Proposal for RASM sub team consideration

Glossary

OA&M agent – High-level management software making decisions for the system.
Managed Chassis Device – Any device in the chassis that is managed (can read data, supports controls)
Management Bridge – Device that connects intra and inter chassis management networks.
Chassis Management Controller – Device (elected or designed to be the management controller for a chassis, managing all non managed devices, performing management functions related to geographic location (ie chassis).

Introduction

This paper is intended to present a suggested implementation of a hardware management layer for the PICMG 3.x effort. There has been lots of good discussion on the philosophies of system management and the requirements needed. In an effort to begin to focus the discussion on the specification, we present this paper which begins to dive more in to the details of how to actually implement a system that addresses the PICMG members and their customers needs. This is not meant to be an exhaustive all inclusive specification, but merely a first take at what we think will be a solution.

We propose a system as described in this paper that leverages current technologies, components and software that meets the needs of the consumers of PICMG 3.x systems.

Intra-chassis management

Physical layer & protocol

We propose using I^2C/IPMB as the physical layer for connecting all the nodes in the chassis, because it leverages existing work, with good availability of microcontrollers with multiple (2, 3 or more I2C interfaces)

We also propose the use of IPMI over IPMB as the protocol/interface for the hardware management layer.

In order to accommodate the potential number of nodes and varying backplane topologies and meet the reliability needs, we propose standardizing the use of repeaters and isolation logic as outlined below. In addition we propose the standardization of certain "OEM" extensions to IPMI to adapt it to a multi-node and multi-tenant environment.

Bus Loading
I2C is normally limited to 400 pF of capacitance on the bus. This typically limits a system to 10 to 15 nodes on an IPMB. This limit is dependant upon the length of the IPMB traces. However, I2C can be extended to accommodate much higher bus capacitance by using repeaters like the Linear Technologies LTC4300. These repeaters incorporate a rise-time accelerator so that I2C timings are met even with several times the normal bus capacitance.

Noise Immunity
I2C doesn't have the extra noise immunity of differential signaling but this extra noise immunity is not needed within the chassis. The higher speed buses like PCI, CPU local buses, memory buses, etc. generally do not use differential signaling, because they are used within a chassis. Noise immunity would be more important if non-intelligent devices that needed to be hot swapped were to be used on the I2C bus.

Redundant Interfaces
In order to provide a highly available solution redundant interfaces will be used. This need is independent of the physical bus used.

Reliable Interfaces
Redundancy alone will not achieve the high availability requirements. The bus interfaces also need to be reliable.

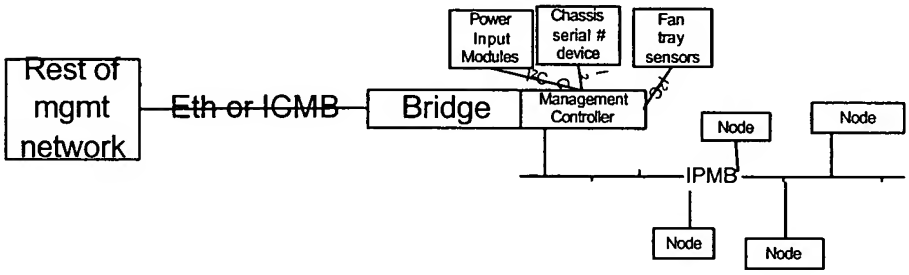
Some people feel that I2C is not a reliable bus. This is not true. This misconception is due to non-intelligent devices that do not correctly implement the I2C interface. This is a moot point for Santa Barbara-based systems, because only intelligent devices will be supported on the management buses. ICMB is intrinsically limited to intelligent devices, so limiting I2C to intelligent devices is not a disadvantage.

Topologies

The intra-chassis management network can take the form of either a bus or a star. For applications that don't need the security or greater isolation capabilities of a dual star, a dual bus can be used.

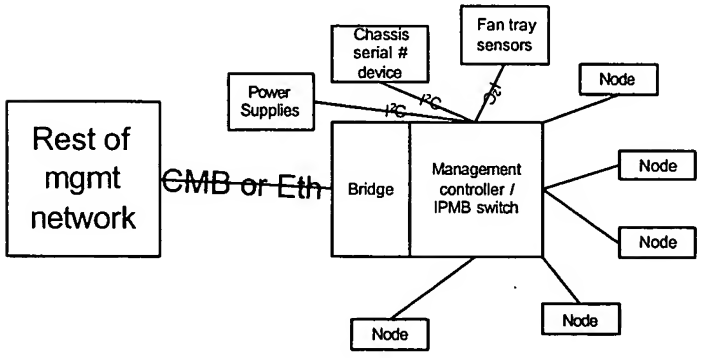
Dual Bus

The dual bus, requires the use of repeaters in order to be able to maintain the physical layer characteristics with 14 slots and intelligent chassis management devices (fan trays, etc.). The shelf management processor provides connections to any non-intelligent chassis devices. A potentially separate entity (could be the shelf management processor) which I refer to as Management Bridge provides the inter-chassis connections.



Dual Star

This topology has all the nodes and managed chassis devices connected via a point to point IPMB connection to each Shelf Management Controller (2 redundant). The management controller could optionally act as an IPMB switch to allow transparent access between the nodes on different spokes of the star. The Shelf Management Controller acts as the bridge to the rest of the management network.



In order to meet the needs for small simple systems as well as larger systems needing more security we propose that we specify both topologies, making sure that nodes work with either topology.

Protocol/Interface

We propose using IPMI v1.5 as the protocol/interface for the hardware management layer. IPMI's broad industry support and existing software, firmware and hardware makes it the best candidate.

Addressing

IPMB does have a smaller address space than ICMB, however, this is not a problem if Ethernet is used for the multi-chassis environments.

ICMB by specification has dynamic (non-deterministic) addressing. A modules address can change at runtime. This can cause interruptions to the manageability of a system. Santa Barbara can specify that modules use a fixed address based on slot number, however, this is an explicit deviation from the ICMB specification.

BMC Determination

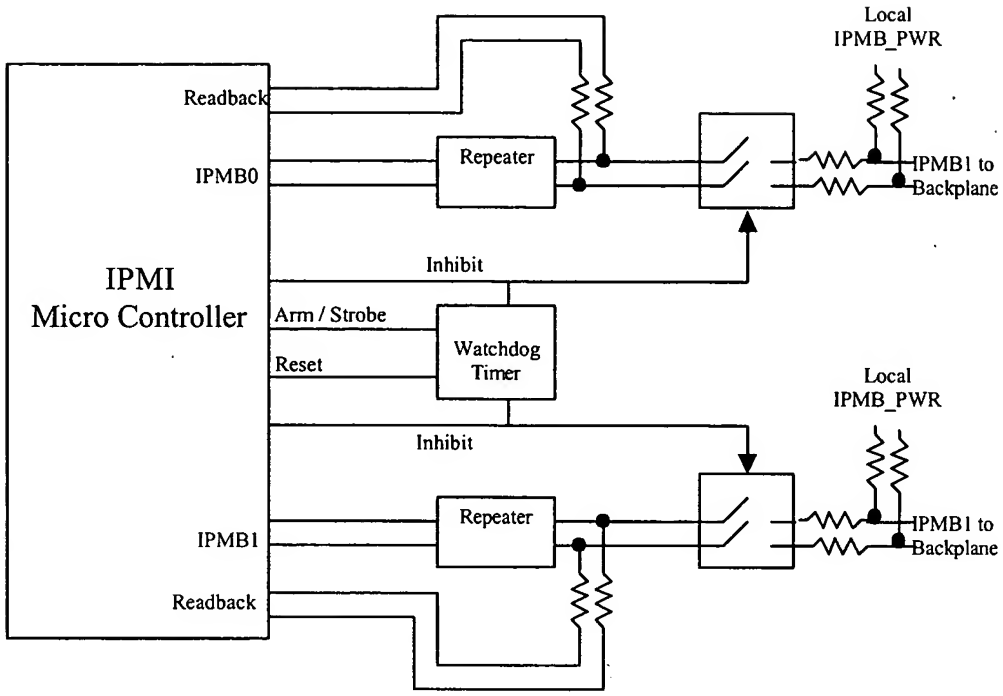
Intel has solved the BMC determination problem. The solution supports determination of the active and standby BMCs in a deterministic fashion. The protocol/algorithm will be published by Intel after approval from the appropriate Intel departments.

Node considerations

In order to ensure interoperability, availability, etc. a lot of work needs to be done on the nodes. We need to make sure that a failure on the nodes don't bring down the management network, that a single failure in the management network doesn't prevent the node from being managed, and that the node supports the functions necessary to manage the node from the management layer.

Although the bus interface is reliable, we need to be able to detect faults, determine the source of the fault to an individual FRU, and isolate the fault to limit the scope of the failure.

The following diagram illustrates how this can be achieved on the redundant I2C interfaces.



Repeaters are used to provide the drive strength needed to overcome the capacitive load for a dual bus topology. Quick switches are used to provide isolation on the IPMBs in the event of a micro controller or bus driver fault. Because IPMI uses an asynchronous, multi-master protocol, the repeater must support I^2C clock-stretching and arbitration in both directions. The repeater must also support standard I2C signal levels (i.e. the Linear Technologies LTC 4300 is suitable, the Philips 9515 and 9516 are not). Two separate repeaters (i.e. physically separate packages) should be used for the two channels to minimize the chance of a single component failure faulting both interfaces. Likewise, separate quick switch devices should also be used.

The watchdog timer is used to detect a micro controller or firmware fault. The watchdog timer must be armed and periodically strobed for the quick switches to allow connection to the backplane.

The micro controller may also inhibit an individual IPMI interface if it detects a fault on an individual interface. The readback lines can be used while the interface is isolated to detect if an onboard error is the root cause of the fault.

The quick switches are not tested, however, this is not a problem. Quick switches have a very high MTBF. Also, when quick switches do fail, they tend to fail either open or closed. Neither of which will fault the bus. They do not tend to short to Vcc or ground, which would fault the bus.

If a quick switch fails in the open state, communication will be lost on the bus and the module will not detect that is the cause of the problem. Intelligent software should be able to identify this fault though, because this would be the only module reporting a communication error (via the other bus).

If a quick switch fails in the closed state, the module may incorrectly determine that it's drivers are not working correctly. This can happen if the controller attempts to isolate from the bus to test its interface. The controller will not be isolated, so other traffic on the bus could look like an interface fault. Although this is an incorrect fault detection, the module is faulty (failed quick switch), so this is not a real problem.

We propose that we require each node to implement isolation and fault detection of the hardware management layer system. We submit the above diagram as a possible implementation.

Node Management Controller

The Node Management Controller needs to support the protocols and interfaces. It needs to be able to have multiple sensors (e.g. temperature sensors, voltage sensors, etc.) The node management controller will support different sensors based upon the board that it is managing. Thus the node management controller needs to support a method of dynamic SDR query so that OA&M entities will be able to read data and exercise controls.

Advantages of having an intelligent entity in the chassis

- ?? Need interface to chassis FRU info
- ?? May have devices in a chassis that need to be managed but don't have a management controller of their own (thus not intelligent and not allowed on management network) such as LCD displays, power input modules, fan trays etc.
- ?? Power sequencing (Quick booting entity, to control boot order of the rest of the chassis)
- ?? Still functional with OS/functional failure of a node
- ?? Can act locally (could be programmed with policies, ie increase fan speed when temp increases)
- ?? Decrease traffic on management network. It has been argued that there won't be much traffic, but if you consider as we move in to the next generation of systems and the need to push for more availability, predictive failure will become important in order to move up the exponential service availability curve. Implementing predictive failure requires take many data points and doing trend analysis. Having a local entity to poll for status when needed can take the burden off a central management entity.

Inter-chassis management

Physical layer & protocol

To connect multiple chassis together on a single management network, we need a physical layer that supports a network that can span multiple frames. There are multiple options for this, but we propose two leading candidates. We need a protocol that can handle addressing of many nodes, that is easy to bridge to internal chassis management networks.

Inter-chassis physical layer

RS-485 is a much better choice for cabling between chassis than I2C. However, another option for managing multiple chassis is Ethernet. The inter-chassis connection is needed for the OA&M agent to communicate with the components that makes up the logical system. If this inter-chassis management network consists of Ethernet, it could allow the OA&M agent to be located outside of the shelves that make up the system, or to provide secure remote management access to the OA&M agent itself.

Option 1: Ethernet

Ethernet equipment and technical knowledge is ubiquitous. It allows for connections between components of 100m, and with routing it can be connected across WANs, allowing an OA&M god entity to manage multiple shelves or systems across multiple geographic locations. The cabling is cheap and easy to manage. The Shelf Management controller could simply wrap IPMB in packets or it could talk higher level IP based protocols (e.g. SNMP) depending on the amount of intelligence available. In the case of logical systems being made up of multiple physical chassis, Ethernet allows the chassis to be dynamically and remotely reconfigured into new configurations.

Option 2: ICMB

Less intelligence (also no NIC, only a transceiver) is needed to bridge from IPMB to ICMB. RS485 is multi-drop and multi-master. Cabling isn't as widely available as Ethernet, but there is no hub/switch needed (which implies reconfiguration requiring a technician being dispatched to a remote facility to change the cabling).

Topologies

The inter-chassis management network would need to be able to connect at least n chassis. (n is undefined here. A reasonable number might be 16?. This number can be arrived at by 4 shelves per frame and large systems consisting of say 4 frames)

The topology must provide redundancy, and must be easy to setup and service.

- ?? A dual star topology would require 2N cables, 2 hubs/switches (each with N hub/switch ports.)
- ?? A dual ring (daisy chain) topology would require 2N cables, 0 hubs/switches. It will handle a single disconnection, but multiple disconnections can break the network.
- ?? A multi-drop connection would require two cables.
- ?? A mesh is not viable because it would require $O(N^2)$ cables and complex bridging between inter and intra chassis management.

Chassis connection considerations

If RS485 were used as the physical layer between chassis, it would need termination to run at speeds greater than 9.6 or 19.2kbps. This termination would require active components, but MAY be enabled on each connection in a "pay as you go" model (Intel has not investigated this possibility). For Ethernet, each chassis would need Ethernet adapters connected to an intelligent entity on the management network such as the shelf management controller.

Security

As Kirk Bresniker of HP has written in his white paper, security is very important for some applications such as multi-tenant chassis, system with nodes on a public IP network, and systems performing sensitive operations such as e-Commerce.
Some examples of situations we need to prevent:

- ?? Malicious hacker entering through public network, able to access management network through host processor/BMC interface
- ?? Benign co-resident accidentally "manages" the wrong system
- ?? Benign code goes haywire trying to "manage" systems, DOS occurs

The proposal in this document addresses these security concerns by providing the option for the dual star topology. The Shelf management controller at the center of the star ensures that the proper policies can be applied. It can also perform authentication for remote and inter-shelf management. This saves every management microcontroller in the system from having to support resource intensive authentication and encryption, and gives the administrator the advantage of only having to configure access control lists in a few places.

A Bussed topology with in a shelf is adequate for non multi-tenant systems or systems not needing the utmost security. A bussed management network (potentially comprising multiple shelves/systems) authentication would need to be implemented on each microcontroller. The intelligent shelf management controller agent can offload that

processing power to one location, allowing more granular access controls when acting as a proxy for management control and information.

Summary of distributed hierarchical management architecture

?? A Shelf has an entity that acts as a BMC for the shelf, working with the other management controllers to provide control and data for all the components in the shelf.
?? A quasi-intelligent entity (could be above entity or not) acts as bridge between inter and intra chassis management networks.
?? There is significantly more bandwidth available for management, allowing predictive failure and other advanced management practices to occur.
?? Shelf Management Controller can be programmed to act automatically to detect/correct faults, enable failover, and adjust operating parameters.
?? The shelf already contains intelligent entities that could be extended to perform this role.
?? Management can be done centrally or in each shelf or both.
?? This approach provides the security needed for multi-tenant systems, as well as remotely managed single shelf systems, leaving the option open for systems that don't need the full security.
?? This architecture is consistent with today's model for IPMI systems.
?? This architecture will leverage investments in microcontrollers, firmware, software and equipment.

This architecture is based upon past work (PICMG 2.9) and actual product implementations. Intel feels that given the schedule we need to meet, that this solution will meet the most number of requirements and goals, at a minimum of cost.

are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

- IPMI -

Intelligent Platform Management Interface Specification

v1.5

Document Revision 1.0
February 21, 2001

Intel Hewlett-Packard NEC Dell

Revision History

Date	Ver	Rev	Modifications
9/16/98	1.0	1.0	IPMI v1.0 Initial release
8/26/99	1.0	1.1	Errata Revision. Incorporated errata from revision 1 or the Errata and Clarifications for the IPMI v1.0 specification.
2/21/01	1.5	1.0	IPMI v1.5 Initial release

Table of Contents

1. Introduction

This document presents the base specifications for the *Intelligent Platform Management Interface* (IPMI) architecture. The IPMI specifications define standardized, abstracted interfaces to the platform management subsystem. IPMI includes the definition of interfaces for extending platform management between board within the main chassis, and between multiple chassis.

The term "platform management" is used to refer to the monitoring and control functions that are built in to the platform hardware and primarily used for the purpose of monitoring the health of the system hardware. This typically includes *monitoring* elements such as system temperatures, voltages, fans, power supplies, bus errors, system physical security, etc. It includes automatic and manually driven *recovery* capabilities such as local or remote system resets and power on/off operations. It includes the *logging* of abnormal or 'out-of-range' conditions for later examination and *alerting* where the platform issues the alert without aid of run-time software. Lastly it includes *inventory* information that can help identify a failed hardware unit.

This document is the main specification for IPMI. It defines the overall architecture, common commands, event formats, data records, and capabilities used across IPMI-based systems and peripheral chassis. This includes the specifications for IPMI management via LAN, serial/modem, PCI Management bus, and the local interface to the platform management. In addition to this document, there is a set of separate supporting specifications:

- The *Intelligent Platform Management Bus* (IPMB) is an I^2C-based bus that provides a standardized interconnection between different boards within a chassis. The IPMB can also serve as a standardized interface for auxiliary or 'emergency' management add-in cards.
- *IPMB v1.0 Address Allocation* documents the different ranges and assignments of addresses on the IPMB.
- The *Intelligent Chassis Management Bus* (ICMB) provides a standardized interface for platform management information and control between chassis. The ICMB is designed so it can be implemented with a device that connects to the IPMB. This allows the ICMB to be implemented as an add-on to systems that have an existing IPMB. See [ICMB] for more information.
- The *Platform Event Trap Format* specification defines the format of SNMP traps used for alerts.
- The *Platform Management FRU Information Storage Definition* defines the format of Field Replaceable Unit information (information such as serial numbers and part numbers for various replaceable boards and other components) accessible in an IPMI-based system.

The implementation of certain aspects of IPMI may require access to other specifications and documents that are not part Refer to the *Reference Documents* section below, for these and other supporting documents.

1.1 Audience

This document is written for engineers and system integrators involved in the design of and interface to platform management hardware, and System Management Software (SMS) developers. Familiarity with microcontrollers, software programming, and PC and Intel server architecture is assumed. For basic and/or supplemental information, refer to the appropriate reference documents.

1.2 Reference Documents

The following documents are companion and supporting specifications for IPMI and associated interfaces:

[ACPI 1.0] *Advanced Configuration and Power Interface Specification*, Revision 1.0b, February 8, 1999. ©1999, Copyright © 1996, 1997, 1998, 1999 Intel Corporation, Microsoft Corporation, Toshiba Corp. http://www.teleport.com/~acpi/

[ACPI 2.0] *Advanced Configuration and Power Interface Specification*, Revision 2.0, July 27, 2000. ©1996, 1997, 1998, 1999, 2000 Compaq Computer Corporation, Intel Corporation, Microsoft Corporation, Phoenix Technologies Ltd., Toshiba Corporation. http://www.teleport.com/~acpi/

[ADDR] *IPMB v1.0 Address Allocation, ©2001 Intel Corporation,* Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. This document specifies the allocation of I^2C addresses on the IPMB. http://developer.intel.com/design/servers/ipmi

[ASF] Alert Standard Forum v1.0 Specification, ©2001, Distributed Management Task Force. http://www.dmtf.org

[CBCP] *Proposal for Callback Control Protocol (CBCP)*, draft-ietf-pppext-callback-cp-02.txt, N. Gidwani, Microsoft, July 19, 1994. As of this writing, the specification is available via the Microsoft Corporation web site: http://www.microsoft.com

[FRU] *Platform Management FRU Information Storage Definition v1.0,* ©1999 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. Provides the field definitions and format of Field Replaceable Unit (FRU) information. http://developer.intel.com/design/servers/ipmi

[I^2C] *The I^2C Bus And How To Use It,* ©1995, Philips Semiconductors. This document provides the timing and electrical specifications for I^2C busses.

[ICMB] *Intelligent Chassis Management Bus Bridge Specification v1.0,* rev. 1.2, © 2000 Intel Corporation. Provides the electrical, transport protocol, and specific command specifications for the ICMB and information on the creation of management controllers that connect to the ICMB. http://developer.intel.com/design/servers/ipmi

[IPMB] *Intelligent Platform Management Bus Communications Protocol Specification v1.0,* ©1998 Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. This document provides the electrical, transport protocol, and specific command specifications for the IPMB. http://developer.intel.com/design/servers/ipmi

[MSVT] *Windows Platform Design Notes, Designing Hardware for the Microsoft® Windows® Family of Operating Systems, Headless Terminal Conventions (VT100+).* ©2000, Microsoft Corporation. http://www.microsoft.com

[PET] *IPMI Platform Event Trap Format Specification v1.0,* ©1998, Intel Corporation, Hewlett-Packard Company, NEC Corporation, and Dell Computer Corporation. This document specifies a common format for SNMP Traps for platform events.

[RFC826] *An Ethernet Address Resolution Protocol -- or -- Converting Network Protocol Addresses to 48.bit Ethernet Address for Transmission on Ethernet Hardware,* David C. Plummer, November 1982

[RFC1319] *RFC 1319, The MD2 Message-Digest Algorithm,* B. Kaliski, RSA Laboratories, April 1992.

[RFC1321] *RFC 1321, The MD5 Message-Digest Algorithm,* R. Rivest, MIT Laboratory for Computer Science and RSA Data Security, Inc. April, 1992.

[RFC1332] *RFC 1332, The PPP Internet Protocol Control Protocol (IPCP),* G. McGreggor, Merit, May 1992.

[RFC1661] *RFC 1661, STD 51, The Point-to-Point Protocol (PPP),* Simpson, W., Editor, Daydreamer, July 1994.

[RFC1662] *RFC 1662, STD 51, PPP in HDLC-like Framing,* Simpson, W., Editor, Daydreamer, July 1994.

[RFC1994] *RFC 1994, PPP Challenge Handshake Authentication Protocol (CHAP),* Simpson, W., Editor, Daydreamer August 1994.

[RFC2153] *RFC 2153, PPP Vendor Extensions*, Simpson, W., Daydreamer, May 1997.

[RFC 2433] *RFC 2433, Microsoft PPP CHAP Extensions*, G. Zorn / S. Cobb, Microsoft Corporation, October 1998

[RFC 2759] *RFC 2759, Microsoft PPP CHAP Extensions, Version 2,* G. Zorn, Microsoft Corporation, January 2000

[SMBIOS] *System Management BIOS Specification, Version 2.3.1,* © 1997, 1999 American Megatrends Inc., Award Software International, Compaq Computer Corporation, Dell Computer Corporation, Hewlett-Packard Company, Intel Corporation, International Business Machines Corporation, Phoenix Technologies Limited, and SystemSoft Corporation.

[SMBUS] *System Management Bus (SMBus) Specification, Version 2.0,* ©2000, Duracell Inc., Fujitsu Personal Systems Inc., Intel Corporation, Linear Technology Corporation, Maxim Integrated Products, Mitsubishi Electric Corporation, Moltech Power Systems, PowerSmart Inc., Toshiba Battery Co., Ltd., Unitrode Corporation, USAR Systems.

[TAP] *Telocator Access Protocol version 1.8, February 04, 1997.* ©1997, Personal Communications Industry Association. http://www.pcia.com (As of this writing, the document is found under 'Wireless Resource Center | Protocols', or: http://www.pcia.com/wireres/protocol.htm.) This document specifies a protocol for sending an alphanumeric page by connecting to a paging service via a serial modem.

[TIA-602] *TIA/EIA Standard: Data Transmission Systems and Equipment - Serial Asynchronous Automatic Dialing and Control, TIA/EIA 602, June 1992.* © 1992, Telecommunications Industry Association. Also available from the Electronic Industries Association. This document specifies the dialing protocol commonly used in asynchronous serial modems.

[WFM] *Wired for Management Baseline Version 2.0 Release,* ©1998, Intel Corporation. Attachment A, UUIDs and GUIDs, provides information specifying the formatting of the IPMI Device GUID and FRU GUID and the System Management BIOS (SM BIOS) UUID unique IDs.

1.3 Conventions and Terminology

If not explicitly indicated, bits in figures are numbered with the most significant bit on the left and the least significant bit on the right.

This document uses the following terms and abbreviations:

Table 1-1, Glossary

Term	Definition
Asserted	Active-high (positive true) signals are asserted when in the high electrical state (near power potential). Active-low (negative true) signals are asserted when in the low electrical state (near ground potential).
BMC	Baseboard Management Controller
Bridge	The circuitry that connects one computer bus to another, allowing an agent on one to access the other.
Byte	An 8-bit quantity.
CMOS	In terms of this specification, this describes the PC-AT compatible region of battery-backed 128 bytes of memory, which normally resides on the baseboard.
Deasserted	A signal is deasserted when in the inactive state. Active-low signal names have "_L" appended to the end of the signal mnemonic. Active-high signal names have no "_L" suffix. To reduce confusion when referring to active-high and active-low signals, the terms one/zero, high/low, and true/false are not used when describing signal states.
Diagnostic Interrupt	A non-maskable interrupt or signal for generating diagnostic traces and 'core dumps' from the operating system. Typically NMI on IA-32 systems, and an INIT on Itanium™-based systems.
Dword	Double word is a 32-bit (4 byte) quantity.
EEPROM	Electrically Erasable Programmable Read Only Memory.
EvM	Notation for 'Event Message'. See text for definitions of 'Event Message'.
FPC	Front Panel Controller.
FRB	Fault Resilient Booting. A term used to describe system features and algorithms that improve the likelihood of the detection of, and recovery from, processor failures in a multiprocessor system.
FRU	Field Replaceable Unit. A module or component which will typically be replaced in its entirety as part of a field service repair operation.
Hard Reset	A reset event in the system that initializes all components and invalidates caches.
I^2C	Inter-Integrated Circuit bus. A multi-master, 2-wire, serial bus used as the basis for the Intelligent Platform Management Bus.
ICMB	Intelligent Chassis Management Bus. A serial, differential bus designed for IPMI messaging between host and peripheral chassis. Refer to [ICMB] for more information.
IERR	Internal Error. A signal from the Intel Architecture processors indicating an internal error condition.
IPM	Intelligent Platform Management.
IPMB	Intelligent Platform Management Bus. Name for the architecture, protocol, and implementation of a special bus that interconnects the baseboard and chassis electronics and provides a communications media for system platform management information. The bus is built on I^2C and provides a communications path between 'management controllers' such as the BMC, FPC, HSC, PBC, and PSC.
ISA	Industry Standard Architecture. Name for the basic 'PC-AT' functionality of an Intel Architecture computer system.
KB	1024 bytes
LUN	Logical Unit Number. In the context of the Intelligent Platform Management Bus protocol, this is a sub-address that allows messages to be routed to different 'logical units' that reside behind the same I^2C slave address.

NMI	Non-maskable Interrupt. The highest priority interrupt in the system, after SMI. This interrupt has traditionally been used to notify the operating system fatal system hardware error conditions, such as parity errors and unrecoverable bus errors. It is also used as a Diagnostic Interrupt for generating diagnostic traces and 'core dumps' from the operating system.
MD2	RSA Data Security, Inc. MD2 Message-Digest Algorithm. An algorithm for forming a 128-bit digital signature for a set of input data.
MD5	RSA Data Security, Inc. MD5 Message-Digest Algorithm. An algorithm for forming a 128-bit digital signature for a set of input data. Improved over earlier algorithms such as MD2.
Payload	For this specification, the term 'payload' refers to the information bearing fields of a message. This is separate from those fields and elements that are used to transport the message from one point to another, such as address fields, framing bits, checksums, etc. In some instances, a given field may be both a payload field and a transport field.
PEF	Platform Event Filtering. The name of the collection of IPMI interfaces in the IPMI v1.5 specification that define how an IPMI Event can be compared against 'filter table' entries that, when matched, trigger a selectable action such as a system reset, power off, alert, etc.
PERR	Parity Error. A signal on the PCI bus that indicates a parity error on the bus.
PET	Platform Event Trap. A specific format of SNMP Trap used for system management alerting. Used for IPMI Alerting as well as alerts using the ASF specification. The trap format is defined in the PET specification. See [PET] and [ASF] for more information.
POST	Power On Self Test.
Re-arm	Re-arm, in the context of this document, refers to resetting internal device state that tracks that an event has occurred such that the device will re-check the event condition and re-generate the event if the event condition exists.
SDR	Sensor Data Record. A data record that provides platform management sensor type, locations, event generation, and access information.
SEL	System Event Log. A non-volatile storage area and associated interfaces for storing system platform event information for later retrieval.
SERR	System Error. A signal on the PCI bus that indicates a 'fatal' error on the bus.
SMI	System Management Interrupt.
SMIC	Server Management Interface Chip. Name for one type of system interface to an IPMI Baseboard Management Controller.
SMM	System Management Mode. A special mode of Intel IA-32 processors, entered via an SMI. SMI is the highest priority non-maskable interrupt. The handler code for this interrupt is typically located in a physical memory space that is only accessible while in SMM. This memory region is typically loaded with SMI Handler code by the BIOS during POST.
SMS	System Management Software. Designed to run under the OS.
Soft Reset	A reset event in the system which forces CPUs to execute from the boot address, but does not change the state of any caches or peripheral devices.
Word	A 16-bit quantity.

1.4 Background - Architectural Goals

A number of goals/principles influence the design and implementation of a platform management subsystem that works across multiple platforms. The abstracted, modular, extensible interfaces specified in this document seek to satisfy those goals. The following review is provided to give a framework to assist in the evaluation options in the implementation of this specification.

Provided Layered Management Value

- Provide management value at each level of integration, and have the net value increase as each level is added. I.e. progressing from processor, through chip set, BIOS, baseboard, baseboard with management circuitry, with onboard networking, with intelligent controllers, with managed chassis, system management software, with Remote Management Cards, etc.
- Maintain modularity so that one level does not carry undue cost burden for another. Levels should retain value if separated. Avoid burdening baseboard with cost for chassis-specific management

functions. Avoid building in functions that unduly impede OEMs in providing their own chassis management features.

- Drive intelligence to appropriate level. Don't put complexity in at a level if the next higher level can handle it. I.e. don't do something with a microcontroller if the system's host processor can do it more flexibly and economically.

Plan for Evolution and Re-use

- Architect so existing implementations can be cleanly extended with new functionality, without requiring existing functionality to be re-implemented or redesigned.
- Architect for product families. Avoid 'local optimizations' that benefit one product at the cost of future projects.
- Knowledge and understanding of the architecture is also a valuable commodity. "Re-inventing the wheel" often means retraining the wheel user. Design to preserve the knowledge base of developers, testers, salespersons, and customers by maintaining consistency in architecture and implementation - in hardware implementation, firmware, software, protocols, and interfaces.
- Design for the economic incorporation of changes in the population and implementation of baseboard and remote sensors. Architect to minimize the impact to hardware and software when sensor population or sensor hardware interfaces change.
- Design to maximize 'self configurability' in system management software. I.e. 'Plug 'N Play'. Provide platform resident discovery mechanisms, such as standardized tables, discovery mechanisms, etc. to reduce or eliminate the need to 'customize' system software for different platforms.

Provide Scalability

- Architecture should scale from entry through enterprise and data center class server systems. Architecture should be adaptable from single board and single chassis, through multi-board and multi-chassis systems.
- Apply 'Layered Management Value' concept. Low-end solutions should be a proper functional subset of higher end solutions. Entry solutions should not carry undue burden for higher class systems.

Support OEM Extensibility:

- Provide clean points for OEM extension and integration.
- Provide OEM support in protocol and command specifications. Reserve command numbers, sensor numbers, etc. for OEM extension.

1.5 New for IPMI v1.5

IPMI v1.5 is an extension of the v1.0 specification. IPMI v1.5 also includes learnings, feedback, and features gathered from industry review and experiences deploying IPMI v1.0 enabled systems.

The following goals guided the creation of the IPMI v1.5 specification:

- Help enable "Always Available Manageability" by enabling new access media: LAN, Serial/Modem, and PCI Management Bus.
- Extend "Autonomous Manageability" by defining new automatic alerting and recovery mechanisms.
- Synch-up with and support emergent and existing standards such as PPP, the DMTF Pre-OS Working Group 'Alert Standard Forum' specification, SMBus 2.0, Compact PCI, and the PCI Management Bus.
- Retain as much backward compatibility with IPMI v1.0 as feasible

The following presents a brief summary of some of the more significant additions and enhancements in the IPMI v1.5 specification:

Serial/Modem Messaging and Alerting

The IPMI v1.5 specification defines how IPMI messaging can be accomplished via a direct serial or external modem connection to the BMC. It also includes the specifications for generating alerts, numeric pages, and alphanumeric pages on events.

Serial Port Sharing

This is a capability that works in conjunction with serial/modem messaging and alerting and allows the baseboard serial connector to be shared between use by the BMC and use by the system. This enables coordination between system features such as console redirection and allows the serial connection to be used for run-time applications while still allowing it to be remotely accessed for 'emergency' management.

Boot Options

IPMI v1.5 includes a boot options 'mailbox' that can be used to direct the operation of BIOS and the booting process following a system power up or reset operation.

LAN Messaging and Alerting

The specification defines how IPMI messaging can be accomplished via a LAN connection to the BMC. It also includes the specifications for generating PET format SNMP traps on events.

Extended BMC Messaging 'Channel Model'

IPMI v1.0 introduced a limited capability to use a 'Channel Number' capability that was primarily used for routing messages to the IPMB. IPMI v1.5 expands on the use of channel numbers as a general mechanism for routing messages between different media and organizing the access

Additional Sensor and Event Types

Several new sensor types have been added to IPMI v1.5, including a Slot/Connector sensor for monitoring hot-plug slot status, an ACPI System Power State sensor to support out-of-band monitoring of the system power state, and an enhanced Watchdog sensor that supports events generated by the standardized watchdog timer function.

Platform Event Filtering (PEF)

Platform Event Filtering (PEF) provides a mechanism for configuring the BMC to taking selected actions on event messages that it receives or has internally generated. These actions include operations such as system power-off, system reset, as well as triggering the generation of an alert.

Alert Policies

Alert policies provide a configurable mechanism for configuring and controlling the delivery of an alert to multiple destinations. This enables the creation of 'call down lists' where one alert destination is tried first and then another.

1.6 IPMI Overview

This section presents an overview of IPMI and its main elements and characteristics.

1.6.1 Intelligent Platform Management

The term Intelligent Platform Management refers to autonomous monitoring and recovery features implemented directly in platform management hardware and firmware. The key characteristic of Intelligent Platform Management is that inventory, monitoring, logging, and recovery control functions are available independent of the main processors, BIOS, and operating system. Platform management functions can also be made available when the system is in a powered down state.

Intelligent Platform Management capabilities are a key component in providing enterprise-class management for high-availability systems. Platform status information can be obtained and recovery actions initiated under situations where system management software and normal 'in-band' management mechanisms are unavailable.

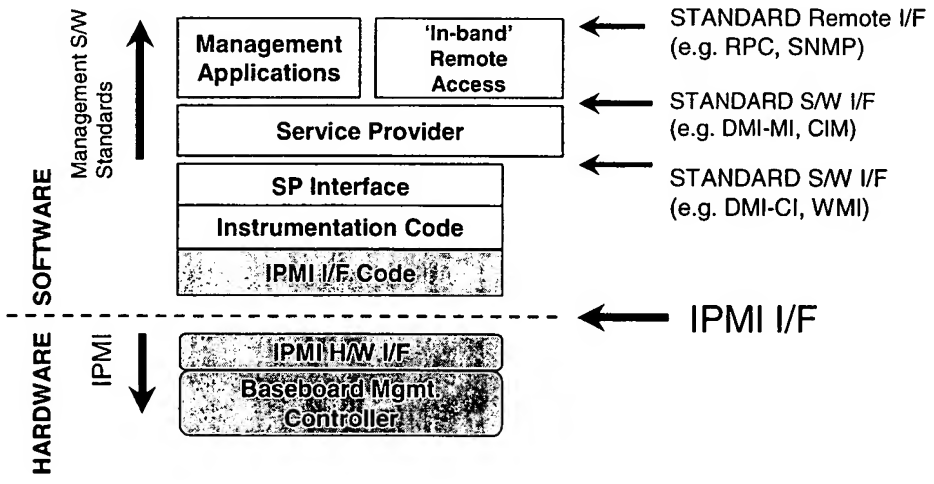
The independent monitoring, logging, and access functions available through IPMI provide a level of manageability built-in to the platform hardware. This can support systems where there is no system management software available for the particular operating system, or the end-user elects not to load or enable the system management software.

1.6.2 IPMI Relationship to other Management Standards

IPMI is best used in conjunction with system management software running under the operating system. This provides an enhanced level of manageability by providing in-band access to the IPMI management information and integrating IPMI with the additional management functions provided by management applications and the OS. System management software and the OS can provide a more sophisticated control, error handling and alerting, than can be directly provided by the platform management subsystem.

IPMI is a hardware level interface specification that is 'management software neutral' providing monitoring and control functions that can be exposed through standard management software interfaces such as DMI, WMI, CIM, SNMP, etc. As a hardware level interface, it sits at the bottom of a typical management software stack, as illustrated in Figure 1-1, below.

Figure 1-1, IPMI and the Management Software Stack

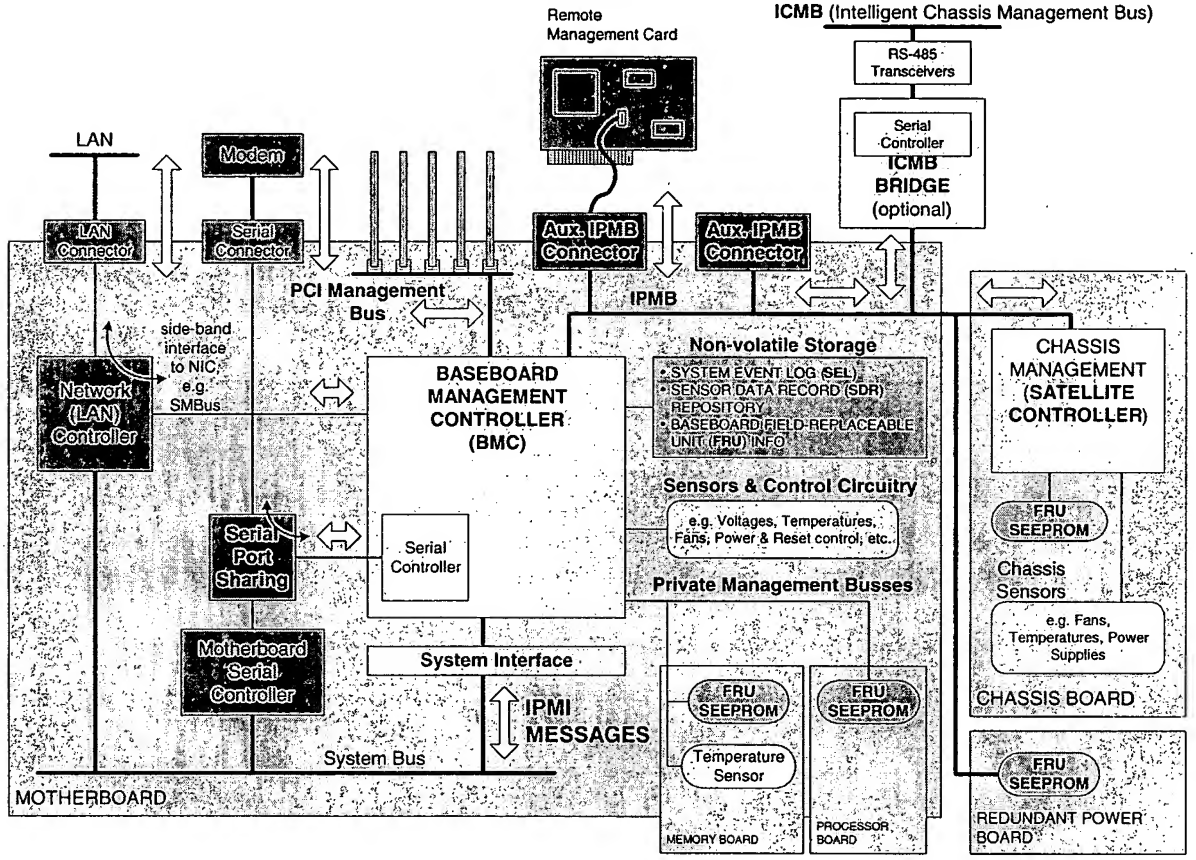


1.6.3 Management Controllers and the IPMB

Error! Reference source not found. shows the main elements of an IPMI implementation. At the heart of the IPMI architecture is a microcontroller called the *Baseboard Management Controller*, or BMC. The BMC provides the intelligence behind Intelligent Platform Management. The BMC manages the interface between system management software and the platform management hardware, provides autonomous monitoring, event logging, and recovery control, and serves as the gateway between system management software and the IPMB and ICMB.

IPMI supports the extension of platform management by connecting additional *management controllers* to the system using the IPMB. The IPMB is an I^2C-based serial bus that is routed between major system modules. It is used for communication to and between management controllers. This provides a standardized way of integrating chassis features with the baseboard. Because the additional management controllers are typically distributed on other boards within the system, away from the 'central' BMC, they are sometimes referred to as *satellite controllers.*

Figure 1-2, IPMI Block Diagram



By standardizing the interconnect, a baseboard can be readily integrated into a variety of chassis that have different management features. IPMI's support for multiple management controllers also means that the architecture is scalable. A complex, multi-board set in an enterprise-class server can use multiple management controllers for monitoring the different subsystems such as redundant power supplies, hot-swap RAID drive slots, expansion I/O, etc., while an entry-level system can have all management functions integrated into the BMC.

IPMI also includes 'low-level' I^2C access commands that can be used to access 'non-intelligent' I^2C devices (devices that don't handle IPMI commands) on the IPMB or *Private Busses* accessed via a management controller. The IPMB can also support SMBus slave devices, with the restriction that the SMB Alert signal is not supported on IPMB, and a controller that implements the IPMB protocol cannot serve as the target for an SMBus *Modified Write Word protocol* transfer from an SMBus slave. Refer to the IPMB and ICMB Specifications (see Reference Documents) for additional information on the IPMB and ICMB.

1.6.4 IPMI Messaging

IPMI uses message-based interfaces for the different interfaces to the platform management subsystem such as IPMB, serial/modem, LAN, ICMB, PCI Management Bus, and the system software-side "System Interface" to the BMC.

All IPMI messages share the same fields in the message 'payload' - regardless of the interface (transport) that they're transferred over. This is represented with the double-ended arrows in **Error! Reference source not found.**. The same core of IPMI messages is available over every IPMI-specified interface, they're just 'wrapped' differently according to the needs of the particular transport. This enables a piece of management software that works on one interface to be converted to use a different interface mainly by changing the underlying 'driver' for the particular transport. This also enables knowledge re-use: A developer that understands the operation of IPMI commands over one interface can readily apply that knowledge to a different IPMI interface.

IPMI messaging uses a request/response protocol. IPMI request messages are commonly referred to as *commands*. The use of a request/response protocol facilitates the transfer of IPMI messages over different transports. It also facilitates multi-master operation on busses like the IPMB and ICMB, allowing messages to be interleaved and multiple management controllers to directly intercommunicate on the bus.

IPMI commands are grouped into functional command sets, using a field called the *Network Function Code*. There are command sets for sensor and event related commands, chassis commands, etc. This functional grouping makes it easier to organize and manage the assignment and allocation of command values.

All IPMI request messages have a *Network Function, command,* and optional *data* fields. All IPMI response messages carry *Network Function, command,* optional *data,* and a *completion code* field. As inferred earlier, the differences between the different interfaces has to do with the framing and protocols used to transfer this payload. For example, the IPMB protocol adds fields for I^2C and controller addressing, and data integrity checking and handling, whereas the LAN interface adds formatting for sending IPMI messages as LAN packets.

1.6.5 Sensor Model

Access to monitored information, such as temperatures and voltages, fan status, etc., is provided via the IPMI *Sensor Model*. Instead of providing direct access to the monitoring hardware IPMI provides access by abstracted sensor commands, such as the *Get Sensor Reading* command, implemented via a management controller. This approach isolates software from changes in the platform management hardware implementation.

Sensors are classified according to the type of readings they provide and/or the type of events they generate. A sensor can return either an analog or discrete reading. Sensor events can be discrete or threshold-based.

The different event types, sensor types, and monitored entities are represented using numeric codes defined in the IPMI specification. IPMI avoids reliance on strings for management information. Using numeric codes facilitates internationalization, automated handling by higher level software, and reduces management controller code and data space requirements.

1.6.6 System Event Log and Event Messages

The same approach is applied to the generation and control of platform events. The BMC provides a centralized, non-volatile *System Event Log,* or SEL. Having the SEL and logging functions managed by the BMC helps ensure that 'post-mortem' logging information is available should a failure occur that disables the systems processor(s).

A set of IPMI commands allows the SEL to be read and cleared, and for events to be added to the SEL. The common request message (command) used for adding events to the SEL is referred to as an *Event Message.* Event Messages can be sent to the BMC via the IPMB. This provides the mechanism for satellite controllers to detect events and get them logged into the SEL. The controller that generates an event message to another controller via IPMB is referred to as an *IPMB Event Generator.* The controller that receives event messages is called the *IPMB Event Receiver.*

A generic *Event Receiver* is a controller that accepts a *Platform Event Message* command over whatever media is connected to it, plus internally generated *Event Messages.* The BMC is typically the only generic *Event Receiver* in the system.

Management Controllers that generate Event Messages must know the sensor and event type so it can place that information in the Event Message. This ensures that Event Messages carry important information that can be interpreted without requiring a-priori knowledge of the sensor, or access to the Sensor Data Record for the sensor.

This is often done by 'hardcoding' this relationship into the controller's firmware. However, this approach binds the Sensor Type and Event Type assignment to the generation of event messages. IPMI also includes commands that allow the sensor and event type information to be read from the Sensor Data Record and written into the controller during initialization. This makes it possible to create generic management controllers that do not have to have hard-coded sensor types. For example, a vendor could create a device that provides a number of analog, threshold-based sensors that get assigned as voltage, temperature, or other sensor types according to the type information the system integrator placed in the SDRs for the sensors. An analog input could be assigned as a "+5V" sensor on one system, and a "-12V" sensor on another just by changing the SDRs.

1.6.7 Sensor Data Records & Capabilities Commands

IPMI's extensibility and scalability mean that each platform implementation can have a different population of management controllers and sensors, and different event generation capabilities. The design of IPMI allows system management software to retrieve information from the platform and automatically configure itself to the platform's capabilities. This greatly reduces or eliminates the need for platform-specific configuration of the platform management instrumentation software - enabling the possibility of "Plug and Play" platform-independent instrumentation software.

Information that describes the platform management capabilities is provided via two mechanisms: *capabilities commands* and *Sensor Data Records* (SDRs). Capabilities commands are commands within the IPMI command sets that return fields that provide information on other commands and functions the controller can handle.

Sensor Data Records are data records that contain information about the type and number of sensors in the platform, sensor threshold support, event generation capabilities, and information on what types of readings the sensor provides.

The primary purpose of Sensor Data Records is to describe the sensor configuration to the platform management subsystem to system software. Sensor Data Records *describe* sensors; they do not *instantiate* sensors. For example, adding a new Sensor Data Record does not cause management controller firmware to automatically 'grow' or instantiate a new sensor. But they are used to describe sensors that already exist, and can also be used to tell software to only pay attention to a subset of the available sensors.

Sensor data records have a limited capability to configure *pre-existing* sensors. There is information that an *Initialization Agent* in the BMC to enable or disable sensors and initialize thresholds. This is described more in the following section.

Sensor Data Records also include records describing the number and type of devices that are connected to the system's IPMB, records that describe the location and type of *FRU Devices* (devices that contain field replaceable unit information).

1.6.8 Initialization Agent

SDRs can also hold default threshold values and event generation settings for sensors and management controllers. During system resets, the BMC performs an *initialization agent* function and writes these settings to those sensors that have 'initialization required' field set in their SDR. This eliminates the need for satellite controllers to retain their own non-volatile storage and command interfaces for default settings, and also provides a mechanism to retrigger any events that may have been transmitted before the BMC was ready to accept them. The initialization agent can also be used to assign the Sensor Type to a generic sensor. See section 27.6, *Sensor Initialization Agent*, for details on the initialization agent process.

1.6.9 Sensor Data Record Repository

Sensor Data Records are kept in a single, centralized non-volatile storage area that is managed by the BMC. This storage is called the *Sensor Data Record Repository* (SDR Repository). Implementing the SDR Repository via the BMC provides a mechanism that allows SDRs to be retrieved via 'out-of-band' interfaces, such as the ICMB, a Remote Management Card, or other device connected to the IPMB. Like most Intelligent Platform Management features, this allows SDR information to be obtained independent of the main processors, BIOS, system management software, and the OS.

1.6.10 Private Management Busses

A Private Management Bus (also referred to as Private Bus) is an I^2C bus that is accessed via a management controller by using IPMI commands for low-level I^2C access. Multiple private busses can be implemented behind a single management controller. IPMI supports using private busses as a mechanism for accessing 24C02-compatible SEEPROMs (Serial Electrically Erasable Programmable ROMs) that hold FRU information. Private busses may also be used to provide low-level access interface for other I^2C or SMBus devices, though the IPMI specification does not cover the way such devices would be used. Each management controller can provide up to eight private busses.

1.6.11 FRU Information

The IPMI specifications include support for storing and accessing multiple sets of non-volatile Field Replaceable Unit (FRU) information for different modules in the system. An enterprise-class system will typically have FRU information for each major system board (e.g. processor board, memory board, I/O board, etc.). The FRU data includes information such as serial number, part number, model, and asset tag.

IPMI FRU information can be made accessible via the IPMB and management controllers. The information can be retrieved at any time, independent of the main processor, BIOS, system software, or OS. This allows FRU information to be retrieved via 'out-of-band' interfaces, such as the ICMB, a Remote Management Card, or other device connected to the IPMB. FRU information can even be available when the system is powered down.

These capabilities allow FRU information to be obtained under failure conditions when FRU access mechanisms that rely on the main processor become unavailable. This facilitates the creation of automated remote inventory and service applications.

IPMI does not seek to replace other FRU or inventory data mechanisms, such as those provided by SM BIOS, and PCI Vital Product Data. Rather, IPMI FRU information is typically used to complement that information or to provide information access out-of-band or under 'system down' conditions.

1.6.12 FRU Devices

IPMI provides FRU information in two ways: via a management controller, or via *FRU SEEPROMs*. FRU information that is managed by a management controller is accessed using IPMI commands. This isolates software from direct access to the non-volatile storage device, allowing the hardware implementer to utilize whatever type of non-volatile storage they want.

In order to more economically support providing FRU information on multiple platform modules, IPMI also allows simple 24C02-compatible SEEPROM (Serial Electrically Erasable Programmable ROM) chips to be used for storing FRU information. ('24C02'-type devices are non-volatile storage devices that have a built-in I^2C-compatible interface).

FRU SEEPROMs provide a mechanism for implementing FRU information without requiring a management controller on the field replaceable unit. FRU SEEPROMs can be accessed via a Private Management Bus connected to a management controller, or can be placed directly on the IPMB - See **Error! Reference source not found.**. (Note: depending on the type of device, I^2C addressing places a limit on the number of devices that can be placed directly on the IPMB. Refer to the *IPMB I^2C Address Allocation* specification for more information.)

1.6.13 Entity Association Records

Entity Association Records are a special type of SDR that provides a definition of the relationships among platform entities. For example, an Entity Association can be set up that groups a set of individual power supplies into a redundant power unit. A 'redundancy lost' event on the power unit can then be correlated with the individual power supply failure. Without the Entity Association information, the 'redundancy lost' and 'power supply failed' events would be disjoint events that could only be correlated based on a-priori knowledge of the system.

1.6.14 Linkage between Events and FRU Information

Included in the SDRs is information that indicates which system entity a sensor is monitoring (e.g. a memory board) and also provide a link to the FRU information for the entity. SDRs use a set of codes that specify which controller holds the sensor, the sensor type (e.g. temperature), the particular instance of the sensor (e.g. sensor #2), the sensor's event and reading type (e.g. discrete or threshold-based), the set of events it can generate, and associated bit fields that indicate which specific events a sensor can produce.

The same codes and bit fields directly map to the information that is passed in event messages and logged in the SEL. Thus, a SEL entry can indicate the controller, sensor, sensor type, and event type associated with the event. This information provides a useful level of information by itself - but when combined with SDR information, the event can be correlated to the entity and FRU associated with the event. Correlating an event to the FRU can help guide a service person to the problem area, or even be used to identify the replacement parts they should bring to a site.

1.6.15 Differentiation and Feature Extensibility

Platform management features continue to evolve. While IPMI seeks to provide a standardized interface to cover the majority of platform management needs, explicit provisions have been made throughout IPMI to support OEM differentiation and new features. Special ranges of code values and commands have been reserved to allow OEM sensors, events, and value-added functions to be implemented within the IPMI framework.

1.6.16 System Interfaces

IPMI defines three standardized system interfaces that system software uses for transferring IPMI messages to the BMC. In order to support a variety of microcontrollers, IPMI offers a choice of system interfaces. Using these interfaces is key to enabling cross-platform software. The system interfaces are similar enough so that a single driver can be created that supports all IPMI system interfaces.

The system interface connects to a system bus that can be driven by the main processor(s). The present IPMI system interfaces are I/O mapped. Any system bus that allows the main processor(s) to access the specified I/O locations, and meet the timing specifications, can be used. Thus, an IPMI system interface could be hooked to the ISA-bus, X-bus, or a proprietary bus off the baseboard chip set.

The three IPMI system interfaces are:

Keyboard Controller Style (KCS) — The bit definitions, and operation of the registers follows that used in the Intel 8742 Universal Peripheral Interface microcontroller. The term 'Keyboard Controller Style' reflects the fact that the 8742 interface was used as the legacy keyboard controller interface in PC architecture computer systems. This interface is available built-in to several commercially available microcontrollers. Data is transferred across the KCS interface using a per-byte handshake.

System Management Interface Chip (SMIC) — The SMIC interface provides an alternative when the implementer wishes to use a microcontroller for the BMC that does not have the built-in hardware for a KCS interface. This interface is a three I/O port interface that can be implemented using a simple ASIC, FPGA, or discrete logic devices. It may also be built-in to a custom-designed management controller. Like the KCS interface, a per-byte handshake is also used for transferring data across the SMIC interface.

Block Transfer (BT) — This interface provides a higher performance system interface option. Unlike the KCS and SMIC interfaces, a per-block handshake is used for transferring data across the interface. The BT interface also provides an alternative to using a controller with a built-in KCS interface. The BT interface has three I/O-mapped ports. A typical implementation includes hardware buffers for holding upstream and downstream message blocks. The BT interface can be implemented using an ASIC or FPGA, or may be built-in to a custom-designed management controller.

1.6.17 Other Messaging Interfaces

In addition to the System Interface and IPMB, IPMI messaging can be carried over other interfaces, such as LAN, serial/modem, ICMB, and PCI management bus. IPMI includes a communication infrastructure that supports transferring messages between these interfaces as well as to the BMC.

1.6.18 LAN Interface

The LAN interface specifications define how IPMI messages can be sent to and from the BMC encapsulated in RMCP (Remote Management Control Protocol) packets datagrams. This capability is also referred to as "IPMI over LAN". IPMI also defines the associated LAN-specific configuration interfaces for setting things such as IP addresses other options, as well as commands for discovering IPMI-based systems.

The Distributed Management Task Force (DMTF) specifies the RMCP format. This same packet format is used for non-IPMI messaging via the DMTF's Alert Standard Forum "ASF" specification. Using the RMCP packet

format enables more commonality between management applications that operate in an environment that includes both IPMI-based and ASF-based systems. More information on IPMI and ASF is provided below.

1.6.19 Serial/Modem Interface

The Serial/Modem Interface specifications define how IPMI messages can be sent to and form the BMC via a direct serial or external modem connection. The specification supports three *connection modes* that define the protocol for delivering IPMI messages via serial/modem:

- **Basic Mode:** The IPMI messages are encapsulated with minimal additional framing and escaping for transport over a serial/modem connection. Basic Mode provides the highest performance but requires an 'IPMI-aware' serial application.
- **PPP Mode:** The IPMI messages are encapsulated in the same RMCP format as used for LAN messages, but are delivered via a PPP connection. PPP mode allows remote applications to take advantage of built-in PPP support in the OS for things such as dialing and authentication, and provides the highest commonality with LAN-based software, but at the cost of lower throughput.
- **Terminal Mode:** Terminal Mode defines how IPMI messages can be transferred using printable characters. It also includes a limited number of English ASCII text commands for doing such things as getting a high level system status and causing a system reset or power state change. Terminal mode is lower performance than Basic Mode and more limited in capabilities than both Basic Mode and PPP Mode, but offers a mechanism for those who are transitioning to IPMI and more sophisticated interfaces from a legacy, character-based environment

1.6.20 IPMI and ASF

IPMI and ASF are complementary specifications that can provide platform management in a 'pre-boot' or 'OS absent' environment. IPMI uses a management microcontroller as the main element of the management system, whereas ASF presently focuses on having an 'alert sending device' - typically the network controller - polling devices on the motherboard and autonomously generating alerts. As of this writing, ASF's scope primarily covers the way an alert sending device polls sensor device and sends alerts, and the specification of 'LAN' commands for discovering RMCP-based systems and performing emergency reset and power off actions.

This includes the supporting specification of SMBus interfaces to 'ASF Sensor Devices' that can be polled by the alert sending device, the specification of the RMCP packet format, and the specification of SMBus-based commands that can be used to send a 'push' alert via an alert sending device.

While somewhat an oversimplification, ASF may be considered to be scoped for 'desktop/mobile' class systems, and IPMI for 'servers' where the additional IPMI capabilities such as event logging, multiple users, remote authentication, multiple transports, management extension busses, sensor access, etc., are valued. However there are no restrictions in either specification as to the class of system that the specification can be used. I.e. you can use IPMI for desktop and mobile systems and ASF for servers if the level of manageability fits your requirements.

IPMI and ASF share a number of formats, data structures, and enumerations. It is expected that this will continue to grow.

- Shared management packet format: IPMI uses ASF 'RMCP' packet format for delivering IPMI messages over LAN and PPP and ASF messages for LAN discovery. The RMCP format includes a message class explicitly for IPMI use.
- Common LAN Alert Format: Both generate LAN Alerts using the IPMI PET (Platform Event Trap) Specification for SNMP Traps
- Common Flags for boot control: IPMI uses a superset of the boot flags defined in ASF.

- Common enumerations for sensor types and event types: ASF uses the IPMI enumerations for sensor and event types. These values are used in Alerts and ASF Sensor Device Status.
- Common BIOS progress codes: IPMI uses ASF BIOS Error and Progress codes.
- Hardware: IPMI management controllers and ASF alert sending devices can both use ASF Sensor Devices. In an IPMI application these can be place on private management busses and polled by the BMC, they can also be used on the PCI management bus. In an ASF application, the devices would typically always be on the PCI management bus or main SMBus and polled by the Network Controller(s).

1.6.21 LAN Alerting

IPMI supports LAN Alerting in the form of SNMP Traps that follow the Platform Event Trap (PET) format. (Refer to [PET] for more information.) SNMP Traps are typically sent as unreliable datagrams. However, IPMI includes a PET Acknowledge and retry options that allows an IPMI-aware remote application to provide a positive acknowledge that the trap was received.

1.6.22 Serial/Modem Alerting and Paging

The IPMI specification supports several options for alerting over a serial/modem connection:

- PPP Alert: The BMC connects to a remote LAN via PPP and delivers a PET trap to a specified IP address.
- Dial Page: Sending a numeric page by using an external modem to generate 'touch tones'.
- TAP Page: The BMC connects to a TAP 1.8 paging service and delivers an alphanumeric page.

1.6.23 Platform Event Filtering (PEF)

Platform Event Filtering (PEF) provides a mechanism for configuring the BMC to taking selected actions on event messages that it receives or has internally generated. These actions include operations such as system power-off, system reset, as well as triggering the generation of an alert.

The BMC maintains an *event filter* table that is used to select which events trigger an action. Each time the BMC receives an event message (either externally or internally generated) it compares the event data against the entries in the event filter table. The BMC scans all entries in the table and collects a set of actions to be performed as determined by the entries that were matched.

1.6.24 Call Down Lists and Alert Policies

The IPMI specification allows an implementation to support configurable alert policies that determine how an alert will be processed. These can be used to create a 'call down list' of different destinations that an alert gets sent to. Alert policies can have destinations of different types and on different channels. For example, a policy could be defined to first try to send an alert to LAN address 'A', and if that fails send it to LAN address 'B', and then send a Dial Page via the modem, and if that fails, a TAP page.

IPMI allows the alert destinations to be configured in any order. I.e. you can pick whether an alert goes out via serial/modem first, or via LAN first. The main limitation comes from the number of policy entries that a given implementation supports.

1.6.25 Channel Model, Authentication, Sessions, and Users

IPMI v1.5 incorporates a common communication infrastructure referred to as the 'Channel Model'. This is an extension of the channels that were used as part of messaging in IPMI v1.0.

Channels provide the mechanism for directing the routing of IPMI messages between different media connections to the BMC. A *channel number* identifies a particular connection. For example, 0 is the channel number for the primary IPMB. Up to nine total channels can be supported (the System Interface and primary IPMB, plus seven additional channels with a media type assigned by the implementer.) Channels can thus be used to support multiple IPMB, LAN, Serial, etc., connections to BMC.

Channels can be *session-based* or *session-less.* A *session* is used for two purposes: As a framework for user authentication, and to support multiple IPMI Messaging streams on a single channel. Session-based channels thus have at least one user 'login' and support user and message authentication. Session-less channels do not have users or authentication. LAN and serial/modem channels are examples of session-based, while the System Interface and IPMB are examples of session-less channels.

In order to do IPMI messaging via a session, a session must first be *activated.* The act of activating a session is one of authenticating a particular user. This is accomplished using a 'challenge/response' mechanism, where a challenge is requested using a *Get Session Challenge* command, and the signed challenge returned in an *Activate Session* command.

The specification supports different algorithms for the signature - these are referred to as Authentication Types. Authentication Types include 'none', 'straight password', the MD2 and MD5 message-digest algorithms, etc. For consistency, session-based channels always use the *Get Session Challenge* and *Activate Session* commands to even if Authentication Type is 'none'. (In this case, dummy values are used for the signatures.)

A session has a *Session ID* that is used for tracking the state of a session. The Session ID mechanism allows multiple sessions to be able to be simultaneously supported on a channel.

The message signature, session ID, and other session related information is separate from the actual IPMI message content. Thus, a packet carrying an authenticated IPMI message can be thought of as being comprised of a 'Session Packet' that includes the session-specific fields and carries an IPMI message as its payload.

The concept of *user* is essentially a way to identify a collection of privilege and authentication information. User configuration is done on a *per channel* basis. This means that a given user could have a different password and set of privileges for accessing the BMC via a LAN channel than via a serial channel.

Privilege Levels determine which IPMI commands a given user can execute over a given channel.

Privilege Limits set the maximum privilege level that a user can operate at. A user is configured with a given maximum privilege limit for each channel. In addition there is a *Channel Privilege Limit* that sets the maximum limit for all users on a given channel. The *Channel Privilege Limit* takes precedence over the privilege configured for the user. Thus, a user can operate at a privilege level that is no higher than the lower of the user privilege limit and the channel privilege limit.

1.6.26 Standardized Watchdog Timer

Watchdog Timer capabilities have been commonly deployed in Enterprise-class servers. IPMI provides a standardized interface for a system Watchdog Timer. This timer can be used for BIOS, OS, and OEM applications. The timer can be configured to automatically generate selected actions when it expires, including power off, power cycle, reset, and interrupt. The timer function can also automatically log the expiration event. Setting '0' for the timeout interval allows the timeout action to be initiated immediately. This provides a means for devices on the IPMB, such as Remote Management Cards, to use the Watchdog Timer to initiate emergency reset and other recovery actions dependent on the capability of the timer.

1.6.27 Standardized POH Counter

This is an optional counter to return a counter value proportional to the system operating (S0) power-on hours.

1.6.28 IPMI Hardware Components

IPMI provides very few restrictions on the actual hardware components used to implement the platform management hardware. IPMI seeks to 'standardize the interface, not the implementation'. IPMI was designed so that it can be implemented with 'off-the-shelf' components. Thus, IPMI does not require specific microcontrollers to be used for management controllers, nor special ASICs or proprietary logic devices. As long as the interface, timing and (in the case of IPMB and ICMB) electrical specifications are met, the choice of components is up to the implementer. It is mandatory to implement a system interface that is compatible with one of the three specified system interfaces.

1.7 IPMI and BIOS

The level of interaction between BIOS and IPMI is greatly dependent on the implementation and number of optional capabilities that are to be supported. It is possible to have an IPMI implementation that does not require any BIOS support, other than that required to meet any applicable ACPI or Plug 'N Play requirements for reporting the I/O and/or interrupt resources used by the IPMI system interface.

In some implementations, BIOS may be responsible for the initialization or startup of certain functions in the management controllers, such as setting the initial timestamp time in the SEL and/or SDR devices. BIOS may also perform tests of the platform management hardware and management controllers during POST.

It is recommended that BIOS include provisions for checking and reporting on the basic health of BMC by executing the *Get Self Test Results* command and checking the result.

It's expected that most implementations will provide BIOS features that take advantage of IPMI. For example, it is expected that many implementations will use IPMI to log POST errors, or to log 'system boot' events so that events can be tracked relative to the last boot time. Another expectation is that many systems utilize the IPMI Watchdog Timer function with BIOS.

With IPMI v1.5, the BIOS can share in additional capabilities. For example, IPMI v1.5 defines a new LAN-based interface. The BIOS can help keep the BMC updated with the LAN IP Address assignment. IPMI v1.5 also includes a serial/modem interface with support for a capability called 'serial port sharing' in which the serial controller can be shared between the BMC and BIOS-based serial console redirection. There is also a set of 'boot flags' that BIOS can read to direct its operation following a system management initiated reset, power cycle, or power up.

1.8 System Management Software (SMS)

The Management Controllers, Sensors, SEL information, SDR information, etc., are of limited value without System Management Software to interpret, handle, and present the information. Platform management is only a subset of systems management. System Management Software takes platform management information and links it into other aspects of systems management, such as software management and distribution, alerting, remote console access, etc.

With respect to the platform management architecture and this specification, System Management Software:

- Polls the System Event Log for new Event information, acting on it as appropriate. This may include taking actions such as sending alerts on the network, presenting a local 'pop-up' message, shutting down an application, consolidating the information with other 'system log' data, etc. System Critical Events are primarily communicated to system management software using the System Event Log as a 'mailbox' between the originator of the Event Message and the system management software.
- Manages the System Event Log. The SEL may contain 'critical' event information that should not be lost. Therefore, the SEL device will not automatically clear the SEL if it gets full. This operation is based on the assumption that it is the first events that are most indicative of the root cause of a problem,

and that later events may be 'side-effects' which, if the SEL were implemented as a 'FIFO' could cause the 'root cause' events to get lost. Instead, System Management Software has the responsibility for determining when SEL entries should be cleared. System Management Software can migrate the SEL contents to disk, to the system's event log, or even to remote storage as desired.

- Reads and interprets the SDR Repository information. System Management Software uses this information to determine the sensor population and capabilities. The Sensor Data information can also be presented to provide a description of the system's manageability features.

- 'Polls' sensors. System Management Software takes the SDR information and uses it to access the sensors, either in a polled or 'on demand' basis, to monitor and present the system's current health and status. Note that whenever possible, System Management Software should rely on event generation for detecting error conditions, and avoid the overhead associated with polling. 'Normal' health status does not generally need to be polled, but would be delivered 'on demand'.

- Potential Event Message Source. System Management Software can also send Event Messages to get events added to the System Event Log. This allows SMS to record information that may be required for 'post-mortem analysis' should it become necessary for System Management Software to shut-down, power-cycle, reset, or otherwise 'off line' the system as a response to a system event. The SEL should be reserved for 'critical' hardware-related errors. The majority OS and software errors should not be written to the SEL. Candidate errors for the SEL are errors that block normal 'in-band' management mechanisms.

1.9 SMI Handler

Not all platform management events come through management controllers or from system software. Some events come from baseboard interrupts. This may include platform events such as correctable and uncorrectable ECC errors, critical NMIs (Non-maskable Interrupts) such as PCI PERR (parity error), PCI SERR (system error), bus timeout interrupts, etc. In some implementations, the platform management hardware maps these 'critical interrupts' to the system SMI (System Management Interrupt) signal. The SMI Handler runs, and, as part of handling these critical interrupts, generates an Event Message to cause the event to get logged in the SEL. The SMI Handler can also take autonomous, 'emergency' action, such as powering off or resetting the system, or propagating an NMI to the operating system.

The SMI Handler is typically a routine that is loaded and initialized into a protected area of memory by the BIOS. SMI is the highest priority non-maskable interrupt in the system. When asserted, it switches the processors into 'System Management Mode' (SMM). Upon entry into SMM, the processor state is saved and a memory configuration is entered where the SMI Handler has full access to system memory and I/O space. This allows the SMI Handler to implement its management functions in an OS-independent manner. The key aspect to this being that the SMI Handler code will run even if the OS is 'hung'. This makes it ideal for implementing certain critical and emergency management functions.

The explicit interface and functionality between an SMI Handler and the BMC is implementation dependent and is not covered by this specification. The implementation of system-specific communication interfaces can be aided using the OEM bits and flags in the BMC-system interface commands.

1.10 Overview of Changes from IPMI v1.0

This section assumes familiarity with the IPMI v1.0 specification. If you're new to IPMI, you can skip ahead to the next section. This is not intended to be a complete "to the bit" list of all the changes, but is provided as a guide for understanding what's involved in moving to support IPMI v1.5.

- Most commands that have version numbers had their version numbers rev'd to 51h for IPMI 1.5
- The *Get Device ID* command was extended a couple of optional firmware revision bytes, per NEC request. Just display them as hex-ASCII.
- The IPMI sensor commands are the same as in v1.0, though a small amount of typo corrections and additional clarifications have been made in those sections.
- The IPMI watchdog commands are backward compatible with IPMI v1.0. A previously reserved bit has been defined as a new 'don't stop' bit that allows the watchdog timer to be reconfigured without stopping it.
- The IPMI v1.0 event commands are the same. A couple of new event commands, *Set/Get Last Processed Event* have been added to allow someone using the new IPMI v1.5 PEF capability to set or determine whether or not PEF has pending events to process.
- Sensor Event/Reading Type codes - The POST Error sensor is now called "System Firmware Progress" and includes new offsets for POST errors and progress the follow the DMTF ASF specification. There's also a new 'Management Subsystem Health' sensor type, 28h. Please check the Sensor Event/Reading Type code table for any other changes.
- The SEL Event Record format is the same except that four previously reserved bits now hold a channel number, and the SEL Record version "EvMRev" field goes from 3h to 4h. It's possible that two events would be identical except for the channel number field. Software that handles or displays events should interpret the channel number field in order to differentiate between events coming from different channels.
- The SDRs are the same with the exception of the version number and a field changes to accomodate a necessary fourth bit for the channel number. This change affected SDRs 01h, 02h, 10h, 11h and 12h. The addition of a channel number in the Type 12h SDR caused the two bytes following to get pushed down.
- The Entity Instance value in the Entity Association Record has been split into two ranges: one for 'system relative' IDs and another for 'device relative' IDs. Most implementations will be able to use their existing Entity Instance assignments since the lower range of values are for the 'system relative' Entity Instance values, which map to the IPMI v1.0 definition of the Entity Instance value.
- SDR Type 14h is being deprecated. IPMI 1.5 systems and software should not use SDR Type 14h. Software should use the new *Get Channel Info* command instead.
- The SEL Event Record format is the same except that four previously reserved bits now hold a channel number, and the SEL Record version "EvMRev" field goes from 3h to 4h.
- The IPMB message format remains the same.
- The *Send Message* command is backward compatible with v1.0 with respect to using it to access the IPMB. I.e. you don't need to make any changes to access the IPMB.
- The *Master Write/Read* I^2C has had the "I^2C" dropped from the name. It is now the *Master Write/Read* command. This command is backward compatible with IPMI v1.0. Reserved bits 7:4 in byte one have become a Channel ID, but 0h is when accessing the IPMB or private management busses as in IPMI v1.0. A non-zero channel value would only be used for accessing additional IPMBs or a PCI Management Bus.
- The read/write FRU commands are the same.

2. Logical Management Device Types

The Intelligent Platform Management architecture is comprised of a number of 'logical' management devices. These are implemented by and within the 'physical' system elements such as the management controllers, I^2C bus, system ASICs, etc.

Each 'logical device' type represents the definition of a particular set of mandatory and optional commands. For the purposes of this specification, the logical management devices are:

IPM Device — Intelligent Platform Management device. This represents the 'basic' intelligent device that responds to the platform sensor and event interface messages. All Intelligent Platform Management devices on the IPMB are expected to respond to the mandatory 'IPM Device' commands. These are also referred to as the 'global' commands. Management Controllers that communicate via compatible messages to the system are also considered IPM devices.

Sensor Device — The Sensor Device is a device that provides the command interface to one or more sensors. Sensor Devices provide a set of commands for discovering, configuring and accessing sensors.

SDR Repository Device — The SDR Device is the logical management device that provides the interface to the Sensor Data Records (SDR) for the system. The SDR Device provides a set of commands for storing and retrieving Sensor Data Records.

SEL Device — The SEL Device is the logical management device that provides the interface to the System Event Log for the system. The SEL Device provides a set of commands for managing the System Event Log.

FRU Inventory Device — The FRU Inventory Device provides the interface to a particular module's FRU Inventory (serial number, part number, asset tag, etc.) information. There will typically be one set of FRU Inventory information for each major module in the system. There can be just as many FRU Inventory Devices providing access to that information. The *Primary FRU Inventory Device* for a given management controller is defined as the device that contains the information about the FRU that holds the management controller itself.

Event Receiver Device — The Event Receiver Device accepts and acknowledges Event Request Messages. The normal action for the Event Receiver Device is to then pass the Event Message to the SEL Device for logging. An *IPMB Event Receiver* refers to an Event Receiver that accepts Event Messages from the IPMB.

Event Generator Device — The Event Generator Device represents the functionality that is used to deliver Event Messages to the Event Receiver Device. The Event Generator Device includes commands to allow configuration of Event Message delivery. The term *IPMB Event Generator* refers to the capability to generate an Event Message on the IPMB. The BMC is typically an IPMB Event Receiver, but not an IPMB Event Generator.

Application Device — A physical instantiation of an Intelligent Platform Management device will most likely have some 'device specific' functionality that it implements that falls outside the 'standard' sensor and event functions. This functionality is referred to as the devices 'Application' functionality. Commands that address this functionality are viewed as being handled by an 'Application' logical device.

PEF Device — This logical device represents the functions associated with comparing an event message against a set of selectable 'event filters' and generating a selectable action on a match.

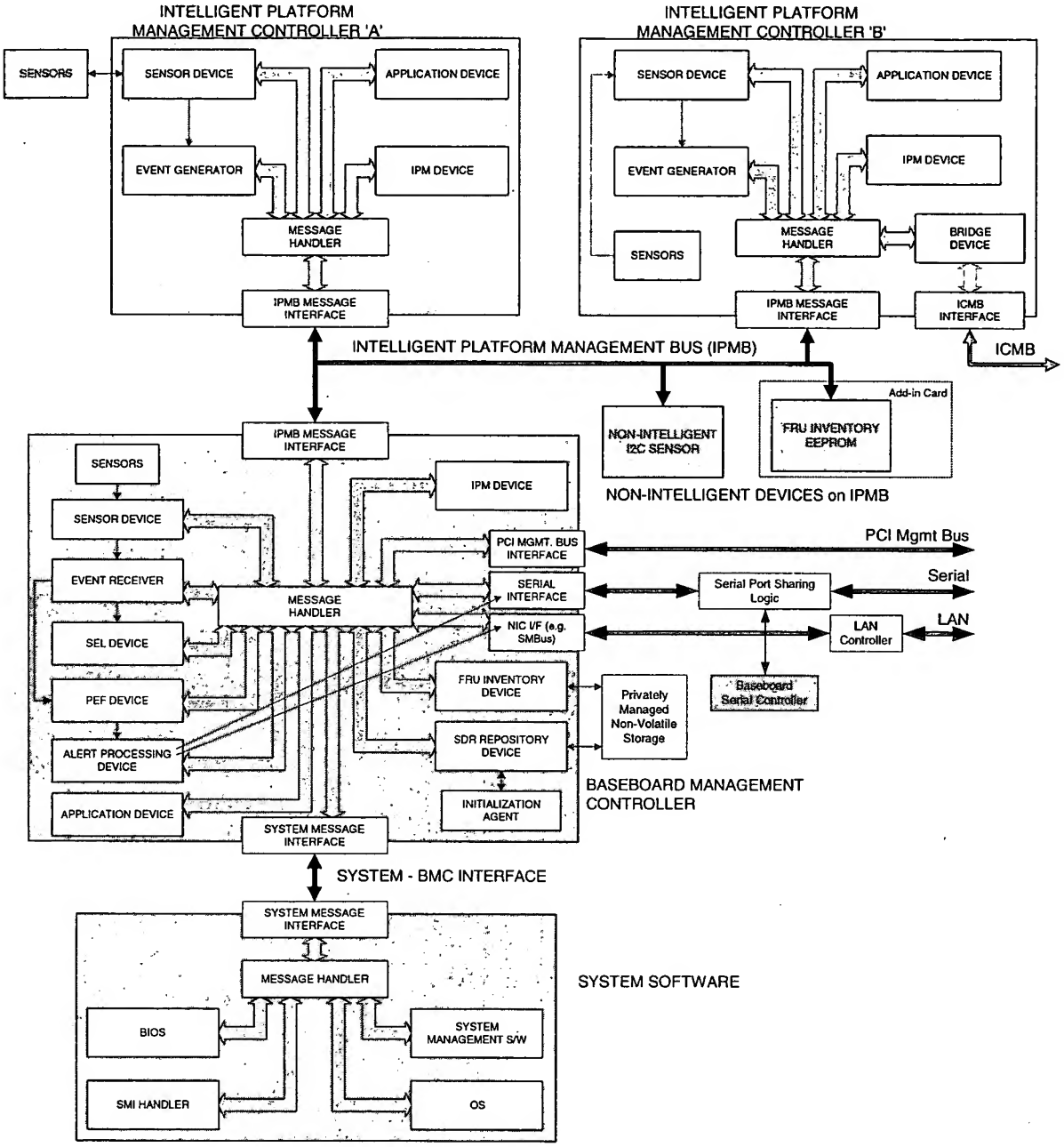
Alert Processing Device — This logical devices represents the functions associated with queuing up and processing alerts, and alert policies that determine which destinations an alert will be sent to.

Chassis Device — This chassis control device represents functions associated with recovery control actions such as power on/off, power cycle, reset, diagnostic interrupt, chassis identification indicator, and system boot.

Message Handler — This logical device represents the functions associated with configuration and operation of message authentication and routing, both internal to the BMC and among the different interfaces to the BMC.

The Intelligent Platform Management Bus can be considered as defining other 'logical' devices as well, such as the 'Bridge' Device for the Intelligent Chassis Management Bus (ICMB). Refer to the *Intelligent Platform Management Bus Protocol Specification* for more information.

Figure 2-1, Intelligent Platform Management Logical Devices



3. Baseboard Management Controller (BMC)

The management architecture can be implemented by centralizing the most common functions into a 'central' management controller in the system. This controller is often called the *Baseboard Management Controller,* or BMC. In some system implementations, the BMC may be the only management controller. The BMC typically provides the following platform management functions:

System Interface — The BMC provides the System Interface to the IPMI-based platform management subsystem. The System Interface is the interface through which system software sends and receives messages to and from the BMC.

Message Handler — The BMC provides functions for routing messages between the different interfaces, including the System Interface, IPMB, serial/modem, LAN, etc. The Message Handler may also be thought of as where shared messaging functions for configuring channel characteristics and user privileges reside.

SEL Interface — The BMC provides the interface to the System Event Log. The BMC allows the SEL to be accessed both from the system side, but also from the Intelligent Platform Management Bus and other external interfaces to the BMC.

Event Generator — The BMC itself will typically be responsible for monitoring and managing the system board. For example, monitoring baseboard temperatures and voltages. As such, the BMC will also be an Event Generator *internally*, sending the Event Messages that it generates internally to its Event Receiver functionality. Note the BMC is not typically and *IPMB Event Generator.* That is, it does not typically issue Event Messages onto the IPMB.

SDR Repository Interface — The BMC will also provide the interface to the SDR (Sensor Data Record) Repository. As with the System Event Log, the BMC allows the records in the SDR Repository to be accessed either via the Intelligent Platform Management Bus or via the system interface.

IPMB Interface — A BMC will typically support an IPMB connection. The IPMB enables the BMC to accept IPMI request messages from other management controllers in the system. The IPMB provides a simple integration point for connecting the 'chassis' management features to the baseboard management. The IPMB can also provide a connection that enables add-in cards to get access to the platform management subsystem.

A BMC that includes IPMB Interface support also provides the capability for system software to send and receive messages to and from the IPMB using the BMC as a kind of communication controller.

IPMB Event Receiver — When an IPMB is implemented, the BMC serves as the primary IPMB Event Receiver for the system. Event Messages can be sent to the BMC from the system or from other controllers the IPMB.

Private Bus Controller — FRU SEEPROMs may be provided on Private Management Busses behind the BMC. The BMC can server as a communication controller that provides access to Private Management Busses and provide access to FRU SEEPROMs and other non-intelligent devices via the *Master Write-Read* command.

FRU Information Interface — The BMC provides access to FRU information for the base system board. The FRU information for the board holding the BMC is obtained by sending *FRU Commands* to the BMC's LUN 00b.

OEM Commands — A BMC implementation can include special support for OEM-unique features and

functions. One way of accomplishing this is by implementing OEM commands through the IPMI messaging interfaces.

Watchdog Timer

A BMC implementation can include special support for OEM-unique features and functions. One way of accomplishing this is by implementing OEM commands through the IPMI messaging interfaces.

In addition, a BMC may implement additional functions for messaging and alerting, including:

Serial/Modem Interface

The BMC can provide a serial/modem interface that allows it to receive IPMI messages over a serial connection to the BMC.

Serial Port Sharing

Serial Port Sharing is a separate capability that works in conjunction with the serial/modem interface. Serial Port Sharing provides a mechanism where the BMC can control logic that allows a single serial connector to be shared between a serial controller on the baseboard and a serial controller for the BMC.

LAN Interface

From the IPMI point-of-view, the interface to the network controller is dedicated to the BMC. That is, there are no special commands for coordinating the sharing of the network controller between system software access and BMC access, as there are with Serial Port Sharing. If the network controller is shared between system software and the BMC, this is generally accomplished via special hardware in the network controller that enable BMC traffic and system traffic to be interleaved.

PCI Management Bus Interface

The BMC can implement a PCI Management Bus Interface that ables the BMC to accept IPMI request messages from add-in cards that plug into a PCI slot. The PCI management bus and IPMB can serve complementary roles. The IPMB providing a mechanism for integrating management functions between baseboard and chassis board functions, while the PCI Management Bus connection can be used to support add-in cards. This division allows the inter-board management communications to be kept separate from add-in card communications.

Platform Event Filtering (PEF)

Platform Event Filtering is an ability for the BMC to perform a configurable action based on an event, by matching the event against a set of 'event filters'. The actions that a BMC can elect to implement include power off, reset, power cycle, generate diagnostic interrupt, and send an alert.

Alert Processing

IPMI v1.5 supports the ability for a BMC to deliver alerts such as SNMP Traps in the Platform Event Trap (PET) format, over media such as LAN and PPP, plus the ability to perform numeric and/or alphanumeric paging via a serial/modem connection. Alert processing includes the ability to support sending alerts to multiple destinations, and to cluster destinations into sets called 'Alert Policies'. Enabling alert policies with PEF makes it possible to configure the system so critical events are delivered to destinations in a 'high priority' alert policy, while non-critical events would go to destinations in a 'low priority' alert policy.

3.1 Required BMC Functions

The following table summarizes the major required and optional functions for an IPMI-conformant BMC.

Table 3-1, Required BMC Functions

Function	M/O	Description
IPM Device	M	The BMC must implement the mandatory IPM Device commands. If an IPMB is provided, the mandatory commands must be accessible from the IPMB unless otherwise noted.
System Interface	M	The implementation must provide BMC access via one of the specified IPMI system interfaces.
SDR Repository	M	The BMC must provide a SDR Repository to hold Sensor, Device Locator, and Entity Association records for all sensors in the platform management subsystem. This does not need to include SDRs for sensors that only generate events. It is recommended that at least 20% additional space is provided for platform management extensions. The SDR Repository must be accessible via the system interface. If an IPMB is provided, the SDR Repository must be readable via that interface as well. SDR update via the IPMB interface is optional. SDR Repository access when the system is powered up or in ACPI 'S1' sleep is mandatory, but access when the system is powered-down or in a >S1 sleep state is optional.
IPMB Interface	O	The IPMB is highly recommended, but optional. The BMC must provide the system interface to the IPMB. If an IPMB is implemented, at least one of the specified IPMB connectors must be provided. Refer to the IPMB Protocol specification for connector definition. In addition the BMC must implement a message channel that allows messages to be sent from the IPMB to the system interface, and vice-versa, and any other mandatory IPMB support functions and commands.
Watchdog Timer	M	The BMC must provide the standardized Watchdog Timer interface, with support for system reset action. Certain functions within the Watchdog Timer are optional. Refer to the sections on the Watchdog Timer for information.
Event Receiver	M	The BMC must implement an Event Receiver function and accept Event Messages via the system interface. If an IPMB is provided, the Event Receiver function must also accept Event Messages from the IPMB. Event Receiver operation while the system is powered up or in ACPI 'S1' sleep is mandatory, but operation when the system is powered down or in a >S1 sleep state is optional.
SEL Interface	M	The BMC must provide a System Event Log interface. The event log must hold at least 16 entries. SEL access must be provided via the system interface. If an IPMB is provided, the SDR Repository must be accessible via that interface as well. SDR Repository access when the system is powered up or in ACPI 'S1' sleep is mandatory, but access when the system is powered-down or in a >S1 sleep state is optional.
FRU Inventory	M (v1.5)	The BMC must provide a logical Primary FRU inventory device, accessible via the *Write-* and *Read FRU Data* commands. The *FRU Inventory Device Info* command must also be supported. It is highly recommended that all other management controllers also provide a Primary FRU inventory device. (This was optional in IPMI v1.0.)
Initialization Agent	M	The initialization agent function is one where the BMC initializes event generation and sensors both internally and on other management controllers according to initialization settings stored in the SDR for the sensor.
Sensors	O	The BMC can provide sensors. A typical server BMC would provide sensors for baseboard temperature, voltage, and chassis intrusion monitoring.
Internal Event	M	The BMC must generate internal events for the Watchdog Timer. It is highly

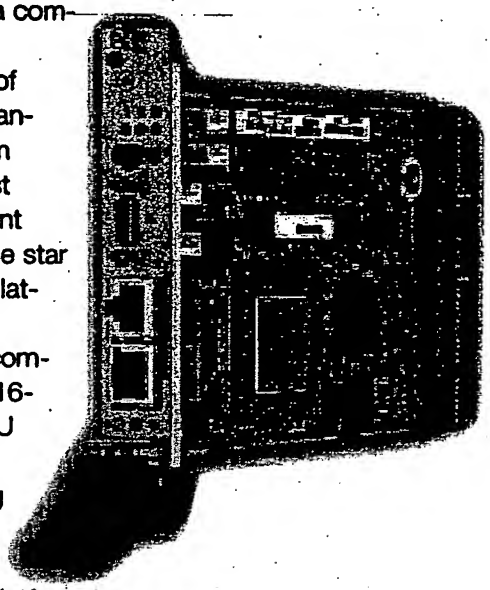
Function	M/O	Description
Generation		recommended that sensors generate events to eliminate the need for system management software to poll sensors, and to provide "post-mortem" failure information in the SEL. Internal event generation for sensors is optional, but highly recommended - particularly for 'environmental' (e.g. temperature and voltage) sensors.
External Event Generation	O	The BMC could be designed to accept the 'Set Event Receiver' command to allow it to be set as an IPMB Event Generator and send its event messages to another management controller. This would primarily be used for development and test purposes.
PCI Management Bus Interface	O	The BMC supports a connection to a PCI Management bus through which the BMC can send and receive IPMI Messages. System software can also access the PCI Management Bus by sending commands to the BMC via the System Interface.
LAN Messaging	O	Ability for the BMC to send and receive IPMI Messaging over LAN
LAN Alerting	O	Ability to send an Alert over the LAN
Serial Messaging	O	Serial messaging is the capability of performing IPMI Messaging over an asynchronous serial connection to the BMC. If Serial Messaging is supported, the following sub-functions apply:
Basic Mode	M	Basic Mode is a type of message framing used for IPMI messaging over a serial connection. Basic Mode support is required if Serial Messaging is supported.
PPP Mode	O	PPP Mode is support for using PPP protocols and framing for IPMI messaging over a serial connection.
Terminal Mode	O	Terminal Mode is a mechanism for IPMI messaging over serial using printable ASCII characters. Terminal mode also supports a limited number of text commands to support legacy 'text based' environments.
Direct Connect Mode	M	Direct Connect Mode is support for IPMI messaging over a serial connection without going through a modem. Direct connect mode is mandatory as part of Serial Messaging.
Modem Connect Mode	O	Direct Connect Mode is support for IPMI messaging over a serial connection through a TIA-602-compatible modem, or via modem circuitry that can work with the IPMI commands defined for modem communication.
Bridging Support	O/M	The ability to transfer IPMI request and response messages between two interfaces connected to the BMC. The following support is required if the corresponding interfaces are supported: • serial/modem ←→ IPMB • serial/modem ←→ System Interface • LAN ←→ IPMB • LAN ←→ System Interface Recommended: • serial/modem ←→ PCI Management Bus • LAN ←→ PCI Management Bus Optional: all other combinations, e.g. serial/modem ←→ LAN
Dial Page	O	Ability to perform a numeric page by dialing. Typically accomplished using an external modem.
PPP Alerting	O	Ability for the BMC to connect to a system Platform Event Filtering and Serial Messaging with PPP Mode are required if PPP Alerting is implemented.

Function	M/O	Description
Callback	O	Callback represents the ability for the BMC to be directed to dial up a selected or pre-configured destination to establish an IPMI Messaging session. Callback requires Serial Messaging with Modem Connect Mode.
Basic Mode Callback	M	Required if Callback is supported. BMC uses Basic Mode for IPMI messaging after connecting to specified destination.
PPP Mode Callback	O	BMC uses PPP Mode for IPMI messaging after connecting to specified destination.
CBCP Callback	O	BMC supports Microsoft CBCP (Callback Control Protocol) for callback. PPP Mode and PPP Mode Callback support are required if CBCP Callback is implemented.
Platform Event Filtering (PEF) and Alert Policies	O/M	Ability for BMC to perform a selectable action on an event. This capability is mandatory if paging or alerting is supported. Certain actions within PEF are optional. Refer to the sections on PEF for information. The Alert action and Alert Policies are mandatory if serial/modem or LAN alerting is supported.

Intel® NetStructure™ ZT 7101 Chassis Management Module

Introduction

Platform management—anticipating and preventing failures, and reducing Mean Time to Repair (MTTR)—is a key requirement for delivering maximum service availability in a computing system. Intel's ZT 7101 3U Chassis Management Module delivers a superb level of security with reliable, comprehensive, IPMI standards-based management (Intelligent Platform Management Interface). Whereas IPMI is most commonly implemented as a bus management topology, Intel has utilized IPMI within a unique star topology, providing enhanced monitoring of platform components.



The ZT 7101 is the central management component for all Intel® NetStructure™ PICMG* 2.16-compliant platforms, including the ZT 5090 4U General Purpose Packet Switched Platform. It uses a standards-based approach, allowing management of third-party IPMI-based products within a NetStructure platform. The ZT 7101 also defines the interface between platform hardware and system software to support higher level management interfaces such as Simple Network Management Protocol (SNMP).

Highlights

- High density 3U x 1-slot CompactPCI* form-factor
- Works within PICMG 2.16- and 2.9-compliant systems
- Enhanced security and reliability utilizing IPMI in a unique star topology
- Provides isolated I2C signals for each slot
- 10/100 Ethernet available at the front or rear panel
- RS-232 (Command Line Interface) available at the front or rear panel
- µDB15 Telco Alarm Interface at the front panel
- Supports multiple management standards including SNMP, IPMI, and Telnet
- Comprehensive monitoring for health, status and component presence
- Out-of-Band, Network Accessible Management Interface
- X-Scale processor design, allowing advanced and automated features to be added with future software releases
- Hot add/Hot swap support for all IPMI-based field-replaceable components
- Individual slot power control for power-on sequencing
- Supports 21 slots when integrated into a custom backplane or chassis

Intel® NetStructure™ ZT 7101 Chassis Management Module

Key Design Elements

Star Topology Security

When used within a NetStructure 2.16-compliant platform, the ZT 7101 provides point-to-point connection to each individual IPMI-based component in a chassis, much like an Ethernet hub or switch in a star topology. The Chassis Management Module (CMM) acts as the primary Baseboard Management Controller (BMC) for the entire chassis, blocking or allowing traffic between any two points of the star, including components and slots. This secure architecture ensures that multiple single board computers with BMCs, housed in the same chassis, do not compete as the primary management controller. It also ensures that no single SBC or application can manage or control other SBCs in the chassis without permission of the CMM.

Comprehensive Management

The CMM may access a comprehensive list of management information and configuration options provided by switched fabric building blocks within a NetStructure platform. It can manage up-to-21 general-purpose nodes, two Ethernet switches, up-to-seven power supplies, up-to-four fan trays (up-to-16 fans total), the chassis sensors, and up-to-two redundant CMMs.

The ZT 7101 CMM queries information from the field-replaceable units (serial number, model number, manufacture date, etc.), and detects presence and performs health monitoring of each component. It also controls the power-up sequencing of each component, and the power-on/off to each slot via BDSEL#. When the thresholds of any IPMI-based component in the chassis are crossed (such as temperature, voltage or performance) or failure occurs, the CMM captures this data, stores it in an event log, sends SNMP traps, and drives the telco alarm and three independent alarm LEDs (Critical, Major, Minor). Users may access previous alerts from the event log.

A Command Line Interface (CLI) allows developers to integrate high-level software management applications with the CMM using a simple IPMI-based library of commands. Administrators may interface with the CLI directly through Telnet to access information about the current state of the system including the sensor values, threshold settings for each sensor, recent events, overall health of the modules, overall health of the chassis, and various power states.

The CMM will integrate with any rack-level management system supporting SNMP traps. It will also generate alerts on health and component thresholds exceeded within the chassis.

High Availability

When two CMM slots are available, two Chassis Management Modules may operate in redundant mode. In this configuration, one CMM is active and the other remains in hot standby mode, ready to take over seamlessly, should the active CMM go out of service. Redundant CMMs are hot-swappable to simplify replacement and minimize service time.

Input/Output

The front panel of the ZT 7101 Chassis Management Module supports the following I/O:

- A 10/100 Ethernet port provides a network exclusively for management, thus avoiding impact to traffic on the primary network. A switch connects Ethernet from the CMM to either the front-panel Ethernet connector or through the packet switched backplane to the Ethernet switch.
- A 15-pin telco connector driven by dry contact relay alarms, and three independent amber LEDs indicating Critical, Major, and Minor events, generate a continuous output until the alarm cut-off button is pressed or turned off via software.
- An RS-232 serial port provides a serial console for access to the Command Line Interface.
- A blue LED indicates hot swap.
- An amber / green / blinking green LED indicates failure / active / standby, respectively.

The CMM also supports pin-out for a rear-panel I/O card.

Warranty

- 2 years

Specifications and Order Options

The most current product specifications and order options are posted on the Web version of this data sheet (www.intel.com/network/csp/products/5090.htm).

Product Information and Sales Support

- www.intel.com
- (805) 541-0488
- ZiatechInfo@Intel.com

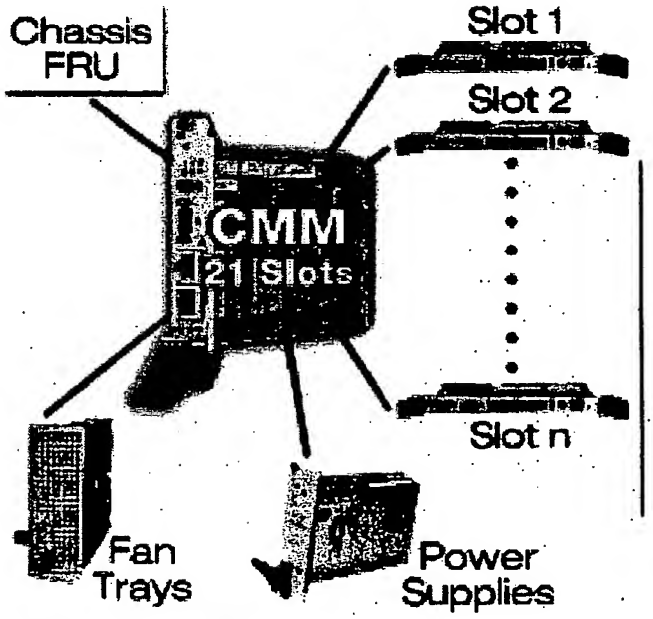


Figure 1: Intel's unique IPMI Star topology

Intel® NetStructure™ ZT 7101 Chassis Management Module

Specifications

Power Requirements	Min.	Typ.	Max.
Supply Voltage, Vcc	4.75V	5.00V	5.25V
Supply Current, Vcc=5.0V	0 mA	-	200 mA

Mechanical

Measures 5.25" x 6.3" (133.4mm x 160mm)
Width: .8" (1 slot-4HP)
Weight: 5.9 oz. (without faceplate and ejector)
Connector: Z-Pack hard metric connectors

Environmental

Operating Temperature: 5 to 55 C
Storage Temperature: -40 to +85 C
Relative Humidity: < 95% at 40 C, non-condensing

Operating System

BlueCat* Linux*

I/O Interface

I/O Interface	Connector Type
RS-232 Serial Port	RJ-45
10/100 Ethernet Port	RJ-45
Telco Alarm Port	Mini DB-15

Order Information

- **ZT 7101:** Chassis Management Module

Please contact Intel Sales Support for information.

7 - Managed Redundant System Slot Operation

This chapter discusses the operation of Managed RSS systems. In Managed mode, extensive use is made of the PICMG 2.9 System Management environment to negotiate and communicate system host/controller status between the RSS hosts. Managed RSS builds on the basic mode signal connections and extends system capabilities by providing the ability to verbosely synchronize and checkpoint system controller operations.

7.1 Introduction

Managed RSS operation extends the Redundant System Slot operations into the high availability environment. This is provided by using the previously described connection signals, with very slight modifications, adding the capabilities of the PICMG 2.9 System Management / IPMI system to the process. The process begins with a determination of a local host s environment using the standard RSS signalling scheme. For a host wishing to use the Managed RSS system, it will read those signals, determine that it may be co-operating with a Managed-RSS capable host, and proceed with host arbitration through the IPMI communications system. At the conclusion of this negotiation, the host will provide host control, if it wins the arbitration, or fall back into a standby or backup role if it lost the arbitration.

During normal operation the hosts may, for any reason, undertake to change their operational status. This is accomplished by a negotiation sequence with their peer, requesting that the peer take on their responsibilities or asking the peer to relinquish the role it currently holds. The system management environment will maintain heartbeats between the two Intelligent Peripheral Managers (IPMs) associated with each host controller. This will be accomplished using the IPMI Watch Dog Timer facility. Should this facility time-out, an error will be declared by the detecting IPM. It will then immediately move to take over the system operation role from the failed host controller in order to maintain system integrity.

The Managed RSS environment provides facilities to control operations of each participating host controller. This includes the ability to notify a single host of the availability of a new system-host capable controller, the establishment of source of the CompactPCI bus clocking/arbitration signals as well as checking and control of the host controllers.

The Managed RSS environment provides redundant communication capabilities between the two system host controllers IPMs by using not only the main system IPMB on the J1/P1 backplane connectors but also using a second IPMB on the J2/P2 connectors. This second IPMB is referred to as the Inter-Host Bus, and is the primary bus for inter-host control communications. The main system management IPMB is used as a backup for this bus in case of a failure of the inter-host bus.

7.2 Arbitration

Managed RSS systems are upwardly compatible with standard RSS systems. In order to determine a system controller s environment and the system capabilities, the Host Arbitration state tables are slightly modified to allow the use of the ALT_SYSEN# signal as an input as well as an output. The modified state machine is shown in Table 7.1. If a system controller wishes to participate in Managed RSS operations, it shall not assert ALT_SYSEN# as an output as it would in basic RSS operations, but rather it should read the signal as an input. This gives the controller the ability to determine the extended system type, and move into Managed role negotiations. Should the host determine that its partner is not capable of, or wishing to participate in, Managed RSS operations, the partner will win the initial role negotiations. The original board may then fall back into basic RSS operations by driving the ALT_SYSEN# signal to the appropriate level or continuing as a peripheral board.

Table 7.1: Host Arbitration

RSS#	SYSEN#	ALT_SYSEN#	Slot Def	Action
HI/Open	HI/Open	X	Peripheral	Enter Satellite Mode
HI/Open	GND	X	Std. System	Enter Host Mode
GND	HI/Open	Drive Low	Basic - RSS	Basic RSS Mode - we are standby host
GND	HI/Open	Read High	Peripheral	Enter Peripheral Mode - No RSS - RSS# in illegal state
GND	HI/Open	Read Low	RSS	Enter Peripheral Mode - Other controller is ZM RSS Host
GND	GND	Drive High	Basic-RSS	Basic RSS Mode - we are active host
GND	GND	Read High	Std. I RSS System	RSS# in illegal state, or ALT_SYSEN# illegal
GND	GND	Read Low	Managed RSS	IPMI Negotiated Mode

7.3 Initial Role Negotiation

When a Managed RSS system-controller capable board is powered up in a system slot, either from a system power-on or a hot-insertion, it will check the RSS signal lines as described in section 7.2. It should also originate an IPMI Device Installed event message to the BMC address on both the inter-host bus as well as the system management bus.. This message serves two purposes. The first is to inform the system BMC and/or current system controller as to the boards new presence. The second is to initiate managed negotiations to allow the newly-installed board to participate in system operations. If the board s IPM determines that the system is not operating in IPMI Negotiated Mode, then it should NOT attempt to take control of the system bus.

The response to the Device Installed message will quickly determine the role of the board. If the I2C data transmission is not byte-acknowledged by the addressee, as per I2C bus rules, it may assume that no system controller is present and it may proceed with asserting control of the CompactPCI bus, and assume the role of BMC for the system as per the PCIMG 2.9 System Management specification. If the I2C transmission *is* acknowledged, the IPM must assume that a system controller has previously been established and continue with negotiation of the board s system role. Table 7.2 shows the data format of the IPMI Device Installed Event Message as derived from Chapters 17 and 30 of the IPMI specification. Note that the address of the board being installed is placed in the header of the event message and is not duplicated in the message text.

Table 7.2: IPMI Device Installed Event Message information

Byte	Data	Data Field Information
1	03h	Event Message Revision
2	21h	Sensor Type - Slot /Connector
3	00h	Sensor #
4	6Fh	Event Type + Direction = assertion
5	F1h	Event Data 1
6	00h	PCI slot type
7	GA	Geographic Address of Slot

7.4 Role Swap Negotiation

After the initial role is determined, the roles of the system controller(s) may be re-negotiated and swapped if desired. The procedure for this is for one of the boards to issue a Role Negotiation command to the other board. The second board will acknowledge this command, and generate an answer response. If the first controller agrees with this response, it will acknowledge the response, then both boards will proceed with the role change. In performing the change-over, the boards shall assure that no more than one system controller be in charge of the PCI bus or using the BMC address at one time.The list of the Role Negotiation commands under the PICMG function is:

Table 7.3: Role Negotiation Commands

Command	NetFn	Cmd
Role Negotiation Request	PICMG	00h
Role Negotiation Response	PICMG	01h

The format of the Role Negotiation command is shown in table 7.3.

Table 7.4: Role Negotiation Request Message

	Byte	Data Field Information
Request data	1	Initiator's Present Role
	2	Target's Present Role
	3	Initiator's Requested Role
Response Data	1	Completion Code

The responding device shall reply with a Role Negotiation Response message as in table 7.4

Table 7.5: Role Negotiation Response

	Byte	Data Field Information
Request Data	1	Initiator's Present Role
	2	Target's Present Role
	3	Initiator's Accepted Role
	4	Target's Accepted Role
Response Data	1	Completion Code

If the initiating device agrees with the response, it shall issue a message to the second device using the Role Negotiation Response format from Table 7.4. If the initiating device does NOT agree with the response, it may issue a new Role Negotiation Request, or simply stop the negotiations, aborting the role change. In the role negotiation messages, the following data definitions shall be used:

Table 7.6: Role Definitions

Role	Data
BMC	FFh
Standby-BMC	0Fh
Peripheral	00h

The Negotiated Roles shall be one of the row entries shown in table 7.6. Any entry marked disallowed is not permitted in the outcome of role negotiations and shall be rejected by the receiver.

Table 7.7: Role Negotiation Capabilities

Initiator's role	Target's role	Initiator's request	Initiator's New Role	Target's New Role
BMC	BMC	BMC	Disallowed	Disallowed
BMC	BMC	Standby-BMC	Disallowed	Disallowed
BMC	BMC	Peripheral	Disallowed	Disallowed
BMC	Standby-BMC	BMC	BMC	Standby-BMC
BMC	Standby-BMC	Standby-BMC	Standby-BMC	BMC
BMC	Standby-BMC	Peripheral	Peripheral	BMC
BMC	Peripheral	BMC	BMC	Peripheral
BMC	Peripheral	Standby-BMC	Disallowed	Peripheral
BMC	Peripheral	Peripheral	Disallowed	Disallowed
Standby-BMC	BMC	BMC	BMC	Standby-BMC
Standby-BMC	BMC	Standby-BMC	Standby-BMC	BMC
Standby-BMC	BMC	Peripheral	Peripheral	BMC
Standby-BMC	Standby-BMC	BMC	Disallowed	Disallowed
Standby-BMC	Standby-BMC	Standby-BMC	Disallowed	Disallowed
Standby-BMC	Standby-BMC	Peripheral	Disallowed	Disallowed
Standby-BMC	Peripheral	BMC	Disallowed	Disallowed
Standby-BMC	Peripheral	Standby-BMC	Disallowed	Disallowed
Standby-BMC	Peripheral	Peripheral	Disallowed	Disallowed
Peripheral	BMC	BMC	Disallowed	Disallowed
Peripheral	BMC	Standby-BMC	Standby-BMC	BMC
Peripheral	BMC	Peripheral	Peripheral	BMC
Peripheral	Standby-BMC	BMC	Disallowed	Disallowed
Peripheral	Standby-BMC	Standby-BMC	Disallowed	Disallowed
Peripheral	Standby-BMC	Peripheral	Disallowed	Disallowed
Peripheral	Peripheral	BMC	Disallowed	Disallowed
Peripheral	Peripheral	Standby-BMC	Disallowed	Disallowed
Peripheral	Peripheral	Peripheral	Disallowed	Disallowed

The negotiating devices shall only change their roles if both agree to the new roles and both receive the full negotiation sequences as indicated by receipt of IPMI acknowledgment packets.

7.5 Fault Determination

The cooperating Redundant System Slot Controllers shall maintain periodic operational checks of each other in order to determine a proper operational state of each board. These checks shall make use of the IPMI Watch-dog timer facility by setting up, in the peer controller, a watch-dog timer and periodically resetting the timer. Should the Watch-dog timer in the peer controller expire, this indicates a failure of the "watched" board, and the peer should undertake the necessary procedures to take control of the system. Each controller should also be performing periodic self-checks to make sure that a false fault condition does NOT cause system failover. The IPMI Watch Dog facility contains a warning timeout in addition to the final timeout of the timer. IPMs should make use of the warning timeout to ascertain system state and prepare for system takeover. Such proceedures should include communiations with the "failing" device in order to alert it to the impending failure detection. The peer device should only perform failover actions if the Watch-Dog timer actually reaches its final timeout, indicating the watched IPM cannot reset the watch-dog timer. The peer may then perform whatever action it deems necessary to insure the integrity of the system, including a negotiated takeover, emergency takeover, or peer reset of the failed controller.

System Management Bus (SMBus) Specification

Version 2.0

August 3, 2000

SBS Implementers Forum

Revision No.	Date	Notes
1.0	2/15/95	General Release
1.1	12/11/98	Version 1.1 Release
2.0	8/3/00	Version 2.0 Release

Questions and comments regarding this specification may be forwarded to:
questions@sbs-forum.org

For additional information on Smart Battery System Specifications, visit the SBS Implementer's Forum (SBS-IF) at:
www.sbs-forum.org

Table of Contents

1. Introduction

1.1. Overview

The System Management Bus (SMBus) is a two-wire interface through which various system component chips can communicate with each other and with the rest of the system. It is based on the principles of operation of I^2C^*.

SMBus provides a control bus for system and power management related tasks. A system may use SMBus to pass messages to and from devices instead of tripping individual control lines. Removing the individual control lines reduces pin count. Accepting messages ensures future expandability.

With System Management Bus, a device can provide manufacturer information, tell the system what its model/part number is, save its state for a suspend event, report different types of errors, accept control parameters, and return its status.

1.2. Audience

The target audience for this document includes but is not limited to:

- System designers implementing the System Management Bus Specification in their systems
- VLSI engineers designing chips to connect to the System Management Bus
- Software engineers writing support code for System Management Bus chips

1.3. Scope

This document describes the electrical characteristics, network control conventions and communications protocols used by SMBus devices. These can be thought as existing at the first three layers of the seven-layer OSI network model, that is, the physical, data link and network layers. Functions normally implemented at higher layers of the OSI model are beyond the scope of this document.

The original purpose of the SMBus was to define the communication link between an intelligent battery, a charger for the battery and a microcontroller that communicates with the rest of the system. However, SMBus can also be used to connect a wide variety of devices including power-related devices, system sensors, inventory EEPROMs communications devices and more.

This version of the specification is a superset of previous versions, 1.0 and 1.1. All devices compliant with these previous versions are compliant with this version. Those features new to SMBus with this version of the spec are optional and are appropriate to the new environments enabled by those features. However, if implemented, these new features must be implemented in a manner compliant with this specification.

1.4. Organization of this document

This document is organized to first give the reader an overview of the SMBus and then to delve deeper into its actual working. The major technical discussion appears in three sections that treat the various aspects of the SMBus as they would appear in the first three layers of the OSI reference network model, the physical layer the data link layer and the network layer.

The section on the physical layer sets out SMBus electrical characteristics. The section on the data link layer specifies bit transfers, byte data transfers, arbitration and clock signals. The section on the link layer deals with the general usage model, the concept of addresses in SMBus, the Address Resolution Protocol and the bus data transfer protocol. All aspects of the SMBus proper may be described within the scope of the first three OSI layers.

The SMBus is a multiple attachment bus with no routing capability. Most communication occurs between and involves only two nodes, a master and a slave. Exceptions to this rule occur during and apply to devices that implement the Address Resolution Protocol as well as the Alert Response Address.

Appendixes at the end of this document contain additional information and guides to implementation that the reader may find useful.

1.5. Supporting documents

This specification assumes that the reader is familiar with or has access to the following documents:

- *The I²C-bus and how to use it*, Philips Semiconductors document #98-8080-575-01.
- *ACPI Specification,* Version 1.0b, Intel Corporation, Microsoft Corporation, Toshiba Corp., February 2, 1999 (http://www.teleport.com/~acpi)
- *PCI Local Bus Specification*, revision 2.2, December 18, 1998, (http://www.pcisig.com)
- *SMBus Control Method Interface Specification*, Version 1.0, Smart Battery System Implementers Forum, December 1999

1.6. Definitions of terms

The following terms are defined with respect to this specification and may have other meanings in other contexts. Some of these terms are used throughout the specification while others have meaning only within limited portions. They are defined here so that the reader may be able to find their definitions in one place.

Address Resolution Protocol	A protocol by which SMBus devices with assignable addresses on the bus are enumerated and assigned non-conflicting slave addresses.
Address Resolved flag (AR)	A flag bit or state internal to a device that indicates whether or not the device's slave address has been resolved by the ARP Master.
Address Valid flag (AV)	A flag bit or state internal to a device that indicates whether or not the device's slave address is valid. This bit must be non-volatile for devices that support the Persistent Slave Address.
ARP	Address Resolution Protocol
SMBus ARP Enumerator	An SMBus master that uses a subset of the ARP for the purpose of discovering ARP-capable slave devices and their assigned slave addresses.
ARP Master	The SMBus master (hardware, software or a combination) responsible for executing the ARP and assigning addresses to ARP-capable slave devices. The SMBus Host will usually be the ARP Master but under some circumstances another SMBus master may assume the role. There is only one active ARP Master at any time.
Assigned Slave Address	The address assigned to a slave device by the ARP Master. This address is then used for accesses to the device's core function. Legal values are in the range 0010 000 to 1111 110 with some exceptions (associated with reserved addresses and those consumed by Fixed Slave Address devices).
Bus Master	Any device that initiates SMBus transactions and drives the clock.
Bus Slave	Target of an SMBus transaction which is driven by some master.
Fixed Slave Address	A slave address that cannot be changed. Non-ARP-capable SMBus devices fall into this category. The ARP Master must not assign a used

	(i.e. device is present on the bus) Fixed Slave Address to an ARP-capable device.
Master-receiver	A bus master in an SMBus transaction while it is receiving data from a bus slave during an SMBus transaction
Master-transmitter	A bus master in an SMBus transaction while it is transmitting data onto the bus during an SMBus transaction.
PEC	Packet Error Code
Persistent Slave Address (PSA)	An assigned slave address that is retained through loss of device power.
Reserved	Reserved fields and bits within any of the data structures in this document and in the SMBus ARP data structures in particular are expected to be unused and ignored by software. Reserved bits must written as 0 and read at 0 unless otherwise specified. These bits and fields are reserved for future use and may not be used by OEMs or IHVs.
Repeated Start	A repeated START is a START condition on the SMBus used to switch from write mode to read mode in a combined format protocol (e.g. Byte Read). The repeated START always follows an Acknowledge, and it always indicates that an address phase is beginning.
Self-powered device	A self-powered device is a device powered either by a battery or an external power source, but not by the system of which it is a part and not in any way by the SMBus.
Slave-receiver	A Slave-receiver is a device that acts as a bus slave in an SMBus transaction while it is receiving address, command or other data from a device acting as a bus master in the transaction.
Slave-transmitter	A Slave-transmitter is a device acting as a bus slave in an SMBus transaction while it is transmitting data on the bus in response to a bus master's request
SMBus Device Default Address	The address all ARP-capable slave devices must respond to. After a slave address has been assigned a device must still respond to commands at the SMBus Device Default Address for ARP management. This address is fixed at 1100 001.
SMBus Host Address	The address to which a host must always respond. This address is used by ARP-capable devices to address the host notify command. This address is fixed at 0001 000.
UDID	Unique Device Identifier. A 128-bit value that a device uses during the ARP process to uniquely identify itself.
Used Address Pool	The list of slave addresses known by the ARP Master that are either: • Reserved • Used by non-ARP-capable devices (Fixed Slave Addresses) • Already assigned to ARP-capable devices The ARP Master must not assign addresses from the first two categories to ARP-capable devices.

1.7. Conventions

Throughout this document, SMBus addresses are given in binary format. SMBus addresses are 7 binary bits long and are conventionally expressed as 4 bits followed by 3 bits followed by the letter 'b', for example, 0001 110b. These addresses occupy the high seven bits of an eight-bit field on the bus. The low bit of this field, however, has other semantic meaning that is not part of an SMBus address.

Diagrams of SMBus transaction data structures as they appear on the bus are of the form:

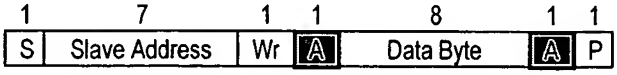


Figure 1-1: generic transaction diagram

In these diagrams, the un-shaded portions are supplied by the bus master of the bus transaction and the shaded portions are driven by the bus slave. The numbers across the top of the transaction diagram indicate the bit widths of each field. In some cases, values will be found below a field in a transaction diagram. When present, this indicates the actual value of the field. The semantics of the various sections of this example transaction are explained in the relevant part of this specification.

The specific SMBus START and STOP conditions, defined in section 4.1.2, are referred to in capital letters.

2. General Characteristics

SMBus is a two-wire bus. Multiple devices, both bus masters and bus slaves, may be connected to an SMBus segment. Generally, a bus master device initiates a bus transfer between it and a single bus slave and provides the clock signals. The one exception to this rule, detailed later, is during initial bus setup when a single master may initiate transactions with multiple slaves simultaneously. A bus slave device can receive data provided by the master or it can provide data to the master.

Only one device may master the bus at any time. Since more than one device may attempt to take control of the bus as a master, SMBus provides an arbitration mechanism that relies on the wired-AND connection of all SMBus device interfaces to the SMBus.

This specification defines two classes of electrical characteristics, low power and high power. The first class, originally defined in the SMBus 1.0 and 1.1 specifications, was designed primarily with Smart Batteries in mind, but could be used with other low-power devices. This version of the specification introduces an alternative higher power set of electrical characteristics. This class is appropriate for use when higher drive capability is required, for example with SMBus devices on PCI add-in cards and for connecting such cards across the PCI connector between each other and to SMBus devices on the system board.

Devices may be powered by the bus V_{DD} or by another power source, V_{Bus}, (as with, for example, Smart Batteries) and will inter-operate as long as they adhere to the SMBus electrical specifications for their class.

The following diagram shows an example implementation of a 5 volt SMBus with devices powered by the bus V_{DD} inter-operating with self-powered devices.

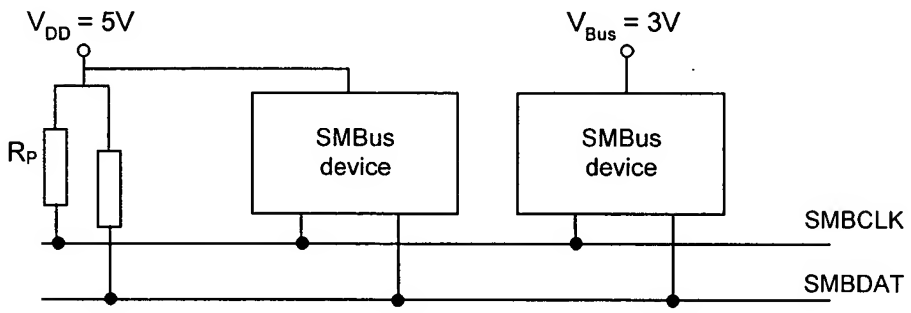


Figure 2-1: SMBus Topology

V_{DD} may be 3 to 5 volts +/- 10% and there may be SMBus devices powered directly by the bus V_{DD}. Both SMBCLK and SMBDAT lines are bi-directional, connected to a positive supply voltage through a pull-up resistor or a current source or other similar circuit. When the bus is free, both lines are high. The output stages of the devices connected to the bus must have an open drain or open collector in order to perform the wired-AND function. Care should be taken in the design of both the input and output stages of SMBus devices, in order not to load the bus when their power plane is turned off, i.e. powered-down devices must provide no leakage path to ground.

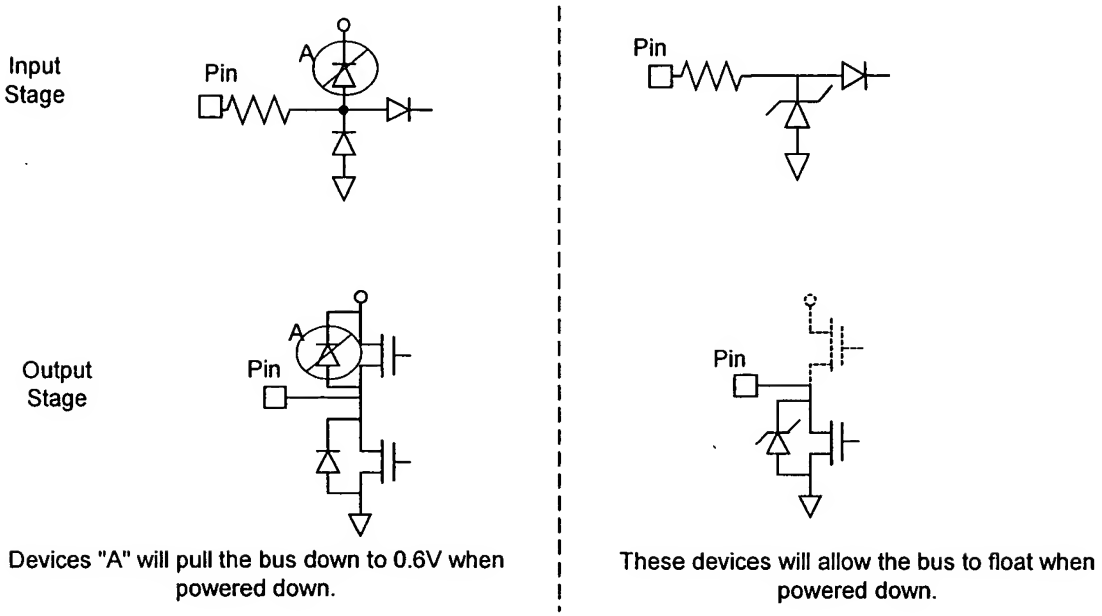


Figure 2-2: Example input and output stages of SMBus devices

A device that wants to place a 'zero' on the bus must drive the bus line to the defined logic low voltage level. In order to place a logic 'one' on the bus the device should release the bus line letting it be pulled high by the bus pull-up circuitry.

The bus lines may be pulled high by a pull-up resistor or by a current source. In cases that involve higher bus capacitance, a more sophisticated circuit may be used that can limit the pull-down sink current while also providing enough current during the low-to-high transition to maintain the rise time specifications of the SMBus.

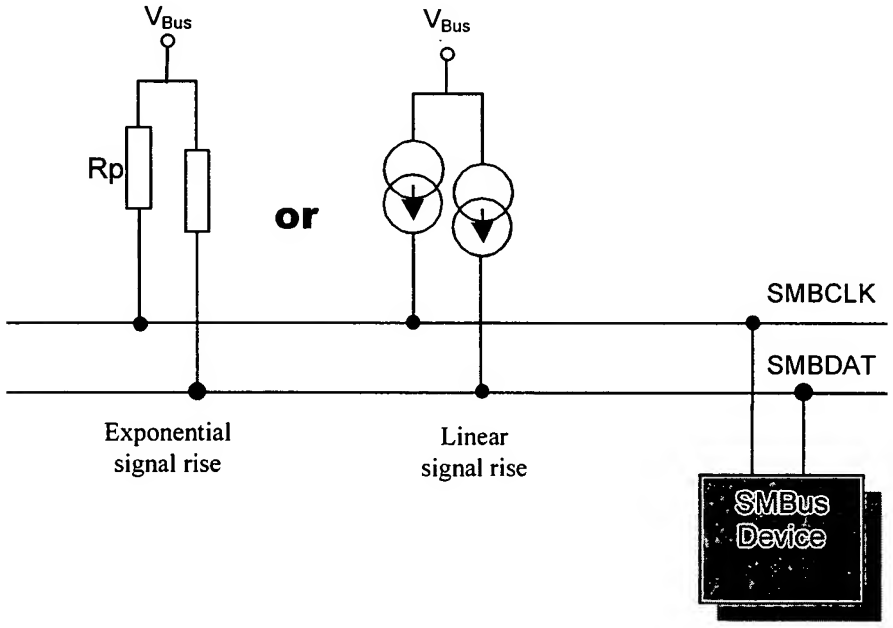


Figure 2-3: SMBus pull-up circuitry

3. Layer 1 – the Physical layer

3.1. Electrical characteristics of SMBus devices – two discrete worlds

SMBus 2.0 is expected to operate in at least two different mutually exclusive environments that have different electrical requirements. In one case, SMBus must operate reliably in the traditional low-power environment of the battery devices that are at its roots. It must also operate reliably in the higher-noise environment of the PCI connector and PCI add-in cards. This specification meets these needs by providing two classes of electrical characteristics as called out below. Most of these specifications deal with voltage levels, noise margins, etc. Of course, many specific items, including general AC specifications, are the same in both environments.

A low-power and a high-power bus may not be directly connected to each other. Low-power devices should not be expected to work on a high-power bus if the device's current sink capability is smaller than 4mA. . However, it is possible to combine a low-power bus segment and a high-power bus segment by connecting the two bus segments through a suitable bridge device.

See Appendix B for ways in which SMBus deviates from I^2C electricals.

3.1.1. SMBus common AC specifications

Both high-power and low-power SMBus electricals share a common set of AC specifications. The diagram below illustrates the various SMBus timings and sets the context for the specifications to follow.

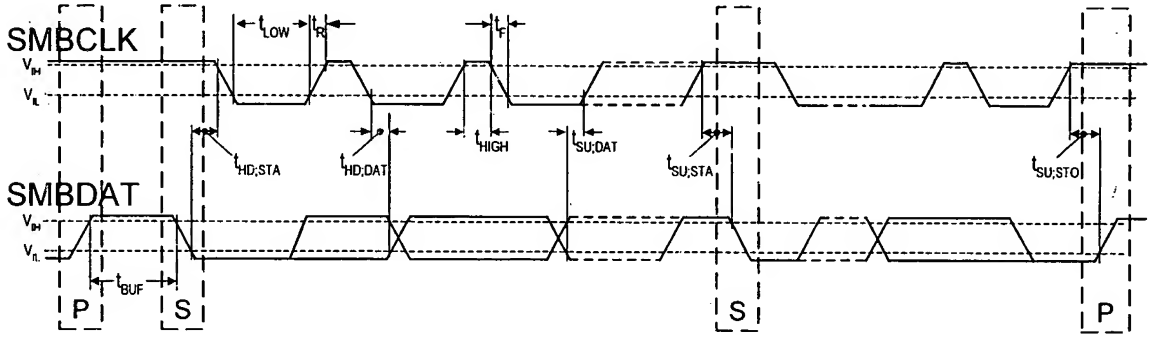


Figure 3-1: SMBus timing measurements

The table below applies to both high and low-power classes of electrical specifications.

Symbol	Parameter	Limits		Units	Comments
		Min	Max		
FSMB	SMBus Operating Frequency	10	100	KHz	See note 1
TBUF	Bus free time between Stop and Start Condition	4.7	-	µs	
THD:STA	Hold time after (Repeated) Start Condition. After this period, the first clock is generated.	4.0	-	µs	
TSU:STA	Repeated Start Condition setup time	4.7	-	µs	
TSU:STO	Stop Condition setup time	4.0	-	µs	
THD:DAT	Data hold time	300	-	ns	
TSU:DAT	Data setup time	250	-	ns	
TTIMEOUT	Detect clock low timeout	25	35	ms	See note 2
TLOW	Clock low period	4.7	-	µs	
THIGH	Clock high period	4.0	50	µs	See note 3

Symbol	Parameter	Limits		Units	Comments
		Min	Max		
TLOW: SEXT	Cumulative clock low extend time (slave device)	-	25	ms	See note 4
TLOW: MEXT	Cumulative clock low extend time (master device)	-	10	ms	See note 5
TF	Clock/Data Fall Time	-	300	ns	See note 6
TR	Clock/Data Rise Time	-	1000	ns	See note 6
TPOR	Time in which a device must be operational after power-on reset		500	ms	See section 3.1.4.2

Table 1: SMBus AC specifications

Note 1: The minimum frequency for synchronizing device clocks is defined in section 4.3.3. A master shall not drive the clock at a frequency below the minimum FSMB. Further, the operating clock frequency shall not be reduced below the minimum value of FSMB due to periodic clock extending by slave devices as defined in section 4.3.3. This limit does not apply to the bus idle condition, and this limit is independent from the TLOW: SEXT and TLOW: MEXT limits.

For example, if the SMBCLK is high for THIGH,MAX, the clock must not be periodically stretched longer than 1/FSMB,MIN – THIGH,MAX. This requirement does not pertain to a device that extends the SMBCLK low for data processing of a received byte, data buffering and so forth for longer than 100us in a non-periodic way.

Note 2: Devices participating in a transfer can abort the transfer in progress and release the bus when any single clock low interval exceeds the value of TTIMEOUT,MIN. After the master in a transaction detects this condition, it must generate a stop condition within or after the current data byte in the transfer process. Devices that have detected this condition must reset their communication and be able to receive a new START condition no later than TTIMEOUT,MAX. Typical device examples include the host controller, and embedded controller and most devices that can master the SMBus. Some simple devices do not contain a clock low drive circuit; this simple kind of device typically may reset its communications port after a start or a stop condition.

A timeout condition can only be ensured if the device that is forcing the timeout holds the SMBCLK low for TTIMEOUT,MAX or longer.

Note 3: THIGH,MAX provides a simple guaranteed method for masters to detect bus idle conditions. A master can assume that the bus is free if it detects that the clock and data signals have been high for greater than THIGH,MAX.

Note 4: TLOW:SEXT is the cumulative time a given slave device is allowed to extend the clock cycles in one message from the initial START to the STOP. It is possible that, another slave device or the master will also extend the clock causing the combined clock low extend time to be greater than TLOW:SEXT. Therefore, this parameter is measured with the slave device as the sole target of a full-speed master.

Note 5: TLOW:MEXT is the cumulative time a master device is allowed to extend its clock cycles within *each byte* of a message as defined from START-to-ACK, ACK-to-ACK, or ACK-to-STOP. It is possible that a slave device or another master will also extend the clock causing the combined clock low time to be greater than TLOW:MEXT on a given byte. Therefore, this parameter is measured with a full speed slave device as the sole target of the master.

Note 6: Rise and fall time is defined as follows:

$$T_R = (V_{ILMAX} - 0.15) \text{ to } (V_{IHMIN} + 0.15)$$

$$T_F = (V_{IHMIN} + 0.15) \text{to } (V_{ILMAX} - 0.15)$$

3.1.1.1. General Timing Conditions

The SMBus is designed to provide a predictable communications link between a system and its devices. However some devices, such as a Smart Battery using a microcontroller to support bus transactions and to maintain battery data, may require more time than might normally be expected. These specifications take such devices into account while maintaining relatively predictable communications. The following are general comments on the SMBus' timing:

- The bus is at 0 KHz when idle. The $F_{SMB,MIN}$ specification is intended to dissuade components from taking too long to complete a transaction.
- Every device must be able to recognize and react to a START condition at the fastest timings in Table 1: SMBus AC specifications, above.

3.1.1.2. Timeouts

Figure 3-2: Timeout measurement intervals illustrates the definition of the timeout intervals, $T_{LOW:SEXT}$ and $T_{LOW:MEXT}$.

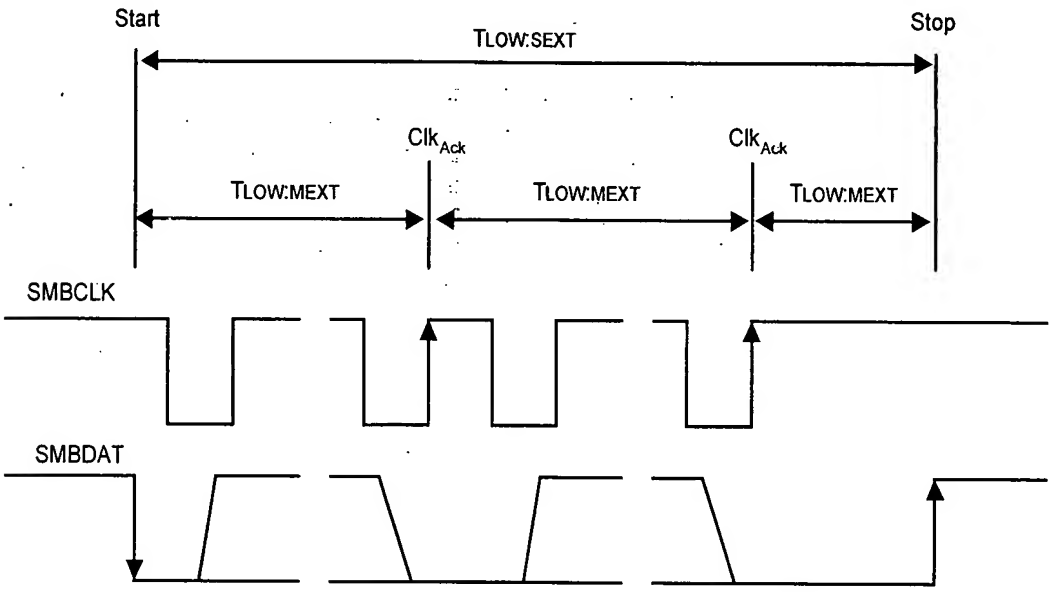


Figure 3-2: Timeout measurement intervals

3.1.1.3. Slave device timeout definitions and conditions

The $T_{TIMEOUT,MIN}$ parameter allows a master or slave to conclude that a defective device is holding the clock low indefinitely or a master is intentionally trying to drive devices off the bus. It is highly recommended that a slave device release the bus (stop driving the bus and let SMBCLK and SMBDAT float high) when it detects any single clock held low longer than $T_{TIMEOUT,MIN}$. Devices that have detected this condition must reset their communication and be able to receive a new START condition in no later than $T_{TIMEOUT,MAX}$.

Slave devices that violate $T_{LOW:SEXT}$ are not conformant with this specification. A Master is allowed to abort the transaction in progress to any slave that violates the $T_{LOW:SEXT}$ or $T_{TIMEOUT,MIN}$ specifications

3.1.1.4. Master device timeout definitions and conditions

TLOW: MEXT is defined as the cumulative time a master device is allowed to extend its clock cycles within one byte in a message as measured from:

START to ACK
ACK to ACK
ACK to STOP.

A system host may not violate TLOW:MEXT except when caused by the combination of its clock extension with the clock extension from a slave device or another master.

A Master is allowed to abort the transaction in progress to any slave that violates the TLOW:SEXT or TTIMEOUT,MIN specifications. This can be accomplished by the Master issuing a STOP condition at the conclusion of the byte transfer in progress.[1]

3.1.2. Low-power DC specifications

3.1.2.1. DC parameters

The System Management Bus is designed to operate over a range of voltages between 3 and 5 Volts +/- 10% (2.7 V to 5.5 V).

Symbol	Parameter	Limits		Units	Comments
		Min	Max		
VIL	Data, Clock Input Low Voltage	-	0.8	V	
VIH	Data, Clock Input High Voltage	2.1	VDD	V	
VOL	Data, Clock Output Low Voltage	-	0.4	V	at IPULLUP, Max
ILEAK	Input Leakage	-	±5	µA	Note 1
IPULLUP	Current through pull-up resistor or current source	100	350	µA	Note 2
VDD	Nominal bus voltage	2.7	5.5	V	3V to 5V ±10%

Table 2: Low power SMBus DC specification

Note 1: Devices must meet this specification whether powered or unpowered. However, a microcontroller acting as an SMBus host may exceed ILEAK by no more than 10 µA.

Note 2: The IPULLUP,MAX specification is determined primarily by the need to accommodate a maximum of 1.1K equivalent series resistor of removable SMBus devices, such as the Smart Battery, while maintaining the VOL,MAX of the bus.

Because of the relatively low pull-up current, the system designer must ensure that the loading on the bus remains within acceptable limits. Additionally, to prevent bus loading, any devices that remain connected to the active bus while unpowered (that is, their Vcc lowered to zero), must also meet the leakage current specification.

[1] A Master should take care when evaluating TLOW:SEXT violation during arbitration since the clock may be held low by multiple slave devices simultaneously. The arbitration interval may be extended for several bytes in the case of devices that respond to commands to the SMBus ARP address. If timeouts are handled at the driver level, the software may need to allow timeouts to be configured or disabled by applications that use the driver in order to support older devices that do not fully meet the SMBus timeout specifications. Devices that implement 'shared' slave addresses may also violate this specification due to combined clock stretching by the different devices sharing the address. TTIMEOUT,MIN, however, does not increase due to combined clock stretching. Therefore, this is a safer timeout parameter for a Master to use when it knows it"s accessing SMBus 2.0 devices.

Systems can be designed today using CMOS components, such as microcontrollers. It is the responsibility of the system designer to ensure that all SMBus components comply with the SMBus timing requirements, and are able to operate within the voltage requirements of the specific system.

Components attached to SMBus may operate at different voltages. Therefore the SMBus cannot assume that all devices will share a common V_{DD}, hence fixed voltage logic levels.

3.1.2.2. SMBus branch circuit model

The following diagram shows the electrical model of the SMBus. A series protection resistor can be used for ESD protection of components that can be hot-plugged to the system, such asa Smart Battery. The Equivalent Series Resistor (ESR) of the device and interconnect must not exceed 1.1K in order to maintain the $V_{OL,MAX}$ of the SMBus low-power specification. This circuit model is equally valid for high-power components discussed in the next section. Due to significantly different bus loading, individual component values will change.

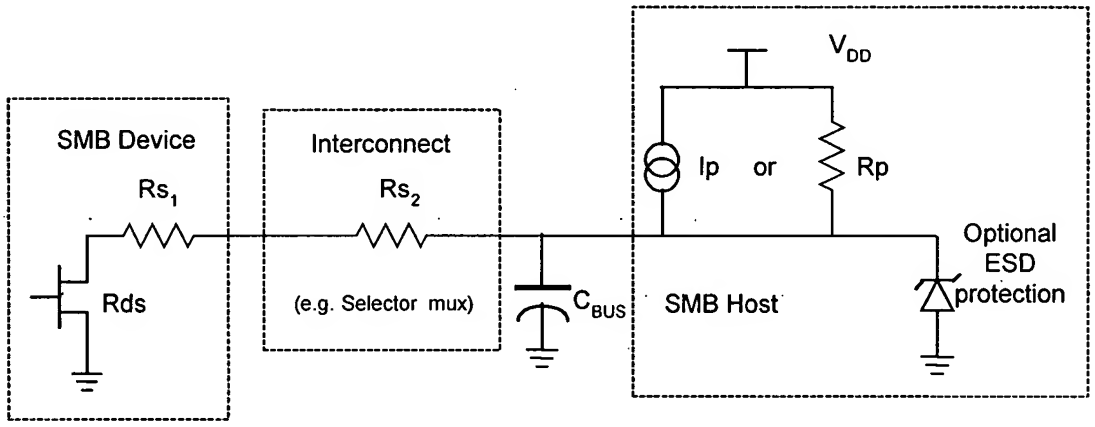


Figure 3-3: SMBus circuit model

The value of the pull-up resistors (Rp) will vary depending on the system's V_{DD} and the bus' actual capacitance. Current sources (Ip) offer better performance but with increased cost.

The optional diode shown in the diagram above is for ESD protection. It may be necessary in systems where a removable SMBus device such as a Smart Battery is used. However, circuits as actually implemented must comply with the previously stated unpowered leakage current specification.

3.1.3. High-Power DC specifications

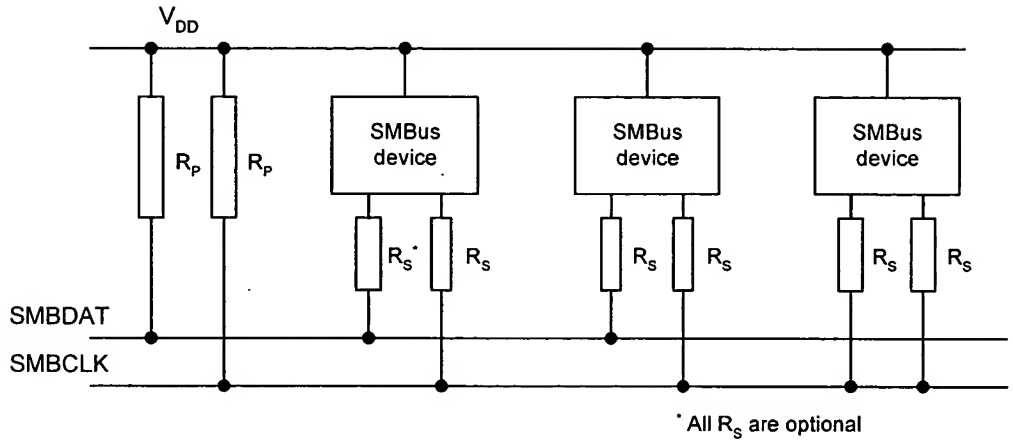


Figure 3-4: SMBus branch with multiple devices attached

High-power SMBus is specified below. These higher power specifications provide the robustness necessary, for example, to allow SMBus to cross the PCI connector, thus allowing SMBus devices on PCI add-in cards to communicate with other devices on both the system board and other PCI add-in cards in the same system. These higher power electrical specifications are an alternative to the lower power specifications stated above and may be used in environments where necessary. Some parameters are explained further in the sections below.

Symbol	Parameter	Limits		Units	Comments
		Min	Max		
V_{IL}	SMBus signal Input low voltage	-	0.8	V	
V_{IH}	SMBus signal Input high voltage	2.1	V_{DD}	V	
V_{OL}	SMBus signal Output low voltage	-	0.4	V	@ I_{PULLUP}
$I_{LEAK-BUS}$	Input Leakage per bus segment		±200	uA	
$I_{LEAK-PIN}$	Input Leakage per device pin		±10	uA	
VDD	Nominal bus voltage	2.7	5.5	V	3V to 5V ±10%
I_{PULLUP}	Current sinking, VOL = 0.4V	4		mA	
C_{BUS}	Capacitive load per bus segment		400	pF	
CI	Capacitance for SMBDAT or SMBCLK pin		10	pF	Recommended
V_{NOISE}	Signal noise immunity from 10 MHz to 100 MHz	300	-	mV p-p	This AC item applies to the high-power DC specification only

Table 3: High-power SMBus DC specification

3.1.3.1. Capacitive load of high-power SMBus lines

Capacitive load for each bus line includes all pin, wire and connector capacitances. The maximum capacitive load affects the selection of the R_P pull-up resistor or the current source in order to meet the rise time specifications of SMBus.

Pin capacitance (CI) is defined as the total capacitive load of one SMBus device as seen in a typical manufacturer's data sheet. The value is a recommended guideline so that two such devices may, for example, be populated on an add-in card.

3.1.3.2. Minimum current sinking requirements for SMBus devices

While SMBus devices used in low-power segments have practically no minimum current sinking requirements due to the low pull-up current specified for low-power segments, devices in high-power segments are required to sink a minimum current of 4 mA while maintaining the $V_{OL,MAX}$ of 0.4 Volts. The requirement for 4 mA sink current determines the minimum value of the pull-up resistor R_P that can be used in SMBus systems.

3.1.4 Additional common low and high-power specifications

3.1.4.1 SMBus 'back powering' considerations

Unpowered devices connected to either a low-power or high-power SMBus segment must provide, either within the device or through the interface circuitry, protection against "back powering" the SMBus. Unpowered devices connected to high-power segments must meet leakage specifications in section 3.1.2.1.

3.1.4.2 Power-on reset

SMBus devices detect a power-on event in one of three ways:

- By detecting that power is being applied to the device,
- By an external reset signal that is being asserted or
- For self-powered or always powered devices, by detecting that the SMBus is active (clock and data lines have gone high after being low for more than 2 1/2 seconds).

An SMBus device must respond to a power-on event by bringing the device into an operational state within TPOR, defined in Table 1, after the device has been supplied power that is within the device's normal operating range. Self-powered or always-powered devices, such as Smart Batteries, are not required to do a complete power-on reset but must be in an operational state within 500 milliseconds after the SMBus becomes active.

4. Layer 2 – the Data Link layer

4.1. Bit transfers

SMBus uses fixed voltage levels to define the logic "ZERO" and logic "ONE" on the bus respectively.

4.1.1. Data validity

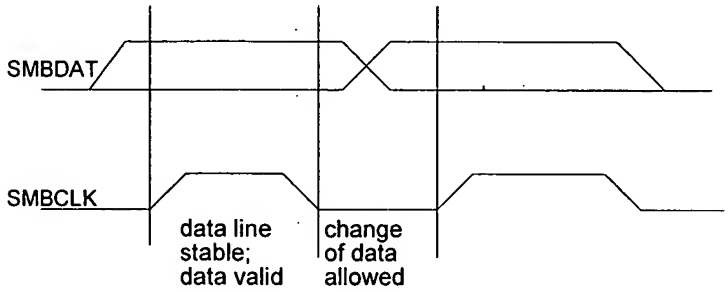


Figure 4-1: Data validity

The data on SMBDAT must be stable during the "HIGH" period of the clock. Data can change state only when SMBCLK is low. Figure 4-1 illustrates the relationships. See figure 3-1 and table 1 for actual specifications.

4.1.2. START and STOP conditions

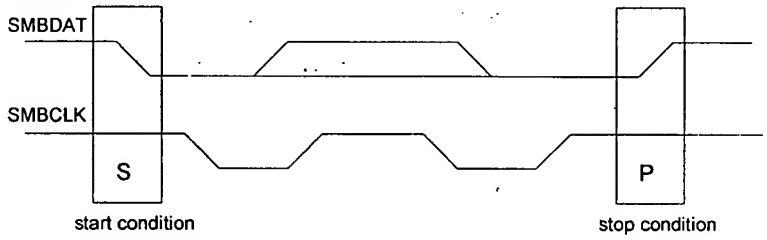


Figure 4-2: START and STOP conditions

Two unique bus situations define a message START and STOP condition.

1. A HIGH to LOW transition of the SMBDAT line while SMBCLK is HIGH indicates a message START condition.
2. A LOW to HIGH transition of the SMBDAT line while SMBCLK is HIGH defines a message STOP condition. START and STOP conditions are always generated by the bus master. After a START condition the bus is considered to be busy. The bus becomes idle again after certain time following a STOP condition or after both the SMBCLK and SMBDAT lines remain high for more than $T_{HIGH:MAX}$.

4.1.3. Bus idle condition

Bus idle is the conditon during which the SMBCLK and SMBDAT lines are both high, without any state transitions, for a time period specified as the minimum of the following:

- T_{BUF} (4.7 µS) from the last detected STOP condition, or
- $T_{HIGH:MAX}$ (50 µS)

The latter timing parameter covers the condition where a master has been dynamically added to the bus and may not have detected a state transition on the SMBCLK or SMBDAT lines. In this case, the master must wait long enough to ensure that a transfer is not currently in progress.

4.2. Data transfers on SMBus

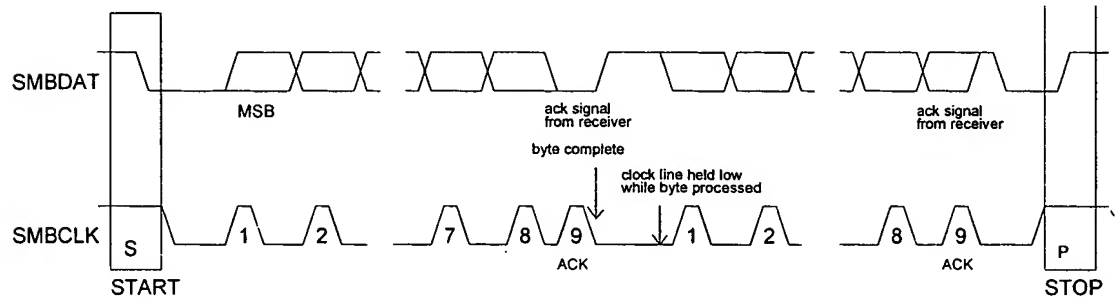


Figure 4-3: SMBus byte format

Every byte consists of 8 bits. Each byte transferred on the bus must be followed by an acknowledge bit. Bytes are transferred with the most significant bit (MSB) first.

The diagram below, figure 4-4 illustrates the positioning of acknowledge (ACK) and not acknowledge (NACK) pulses relative to other data

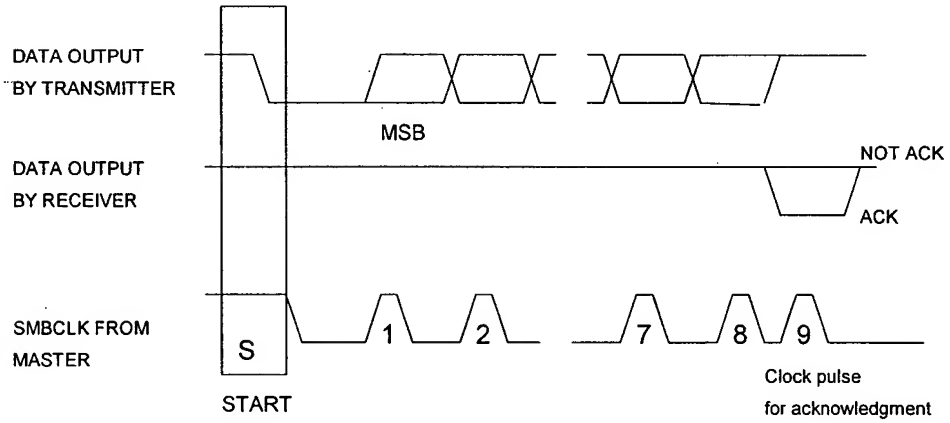


Figure 4-4: ACK and NACK signaling of SMBus

The acknowledge-related clock pulse is generated by the master. The transmitter, master or slave, releases the SMBDAT line (HIGH) during the acknowledge clock cycle. In order to acknowledge a byte, the receiver must pull the SMBDAT line LOW during the HIGH period of the clock pulse according to the SMBus timing specifications. A receiver that wishes to NACK a byte must let the SMBDAT line remain HIGH during the acknowledge clock pulse.

An SMBus device must always acknowledge (ACK) its own address. SMBus uses this signaling to detect the presence of detachable devices on the bus.

An SMBus slave device may decide to NACK a byte other than the address byte in the following situations:

- The slave device is busy performing a real time task, or data requested are not available. The master upon detection of the NACK condition must generate a STOP condition to abort the transfer. Note

that as an alternative, the slave device can extend the clock LOW period within the limits of this specification in order to complete its tasks and continue the transfer.

- The slave device detects an invalid command or invalid data. In this case the slave device must NACK the received byte. The master upon detection of this condition must generate a STOP condition and retry the transaction.
- If a master-receiver is involved in the transaction it must signal the end of data to the slave-transmitter by generating an NACK on the last byte that was clocked out by the slave. The slave-transmitter must release the data line to allow the master to generate a STOP condition.

The latter mechanism is one way for a device to detect whether a slave transmitter implements Packet Error Checking. See section 5.4 for more information on Packet Error Checking.

4.3. Clock generation and arbitration

4.3.1. Synchronization

A situation may occur in which more than one master is trying to place clock signals on the bus at the same time. The resulting bus signal will be the wired AND of all the clock signals provided by the masters.

It is important for the bus integrity that there is a clear definition of the clock, bit by bit for all masters involved during an arbitration process.

A high-to-low transition on the SMBCLK line will cause all devices involved to start counting off their LOW period and start driving SMBCLK low if the device is a master. As soon as a device finishes counting its LOW period it will release the SMBCLK line. Nevertheless, the actual signal on the SMBCLK may not transition to the HIGH state if another master with longer LOW period keeps the SMBCLK line LOW. In this situation the master that released the SMBCLK line will enter the SMBCLK HIGH wait period. When all devices have counted off their LOW period, the SMBCLK line will be released and go HIGH. All devices concerned at this point will start counting their HIGH periods. The first device that completes its HIGH period count will pull the SMBCLK line LOW and the cycle will start again.

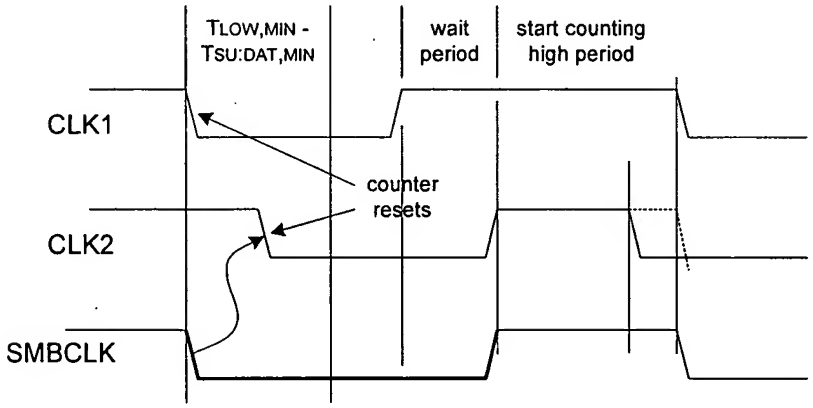


Figure 4-5: SMBus clock synchronization

In Figure 4-5, the interval between the first high-to-low transition of CLK1 and CLK2 must be less than (TLOW:MIN – TSU:DAT). This way, a synchronized clock is provided for all devices, where the SMBCLK

LOW period is determined by the slowest device and the SMBCLK HIGH period is determined by the fastest device.

4.3.2. Arbitration

A master may start a transfer only if the bus is idle. One or more devices may generate a START condition within the minimum hold time ($t_{HOLD:STA}$) resulting in a defined START condition on the bus.

Since the devices that generated the START condition may not be aware that other masters are contending for the bus, arbitration takes place on the SMBDAT line while the SMBCLK is HIGH. A master that transmits a HIGH level, while the other(s) master is transmitting a LOW level on the SMBDAT line loses control of the bus in the arbitration cycle.

The master that lost the arbitration may continue to provide clock pulses until the completion of the byte on which it lost the arbitration. Arbitration in the case of two masters trying to access the same device may continue past the address byte. In this case arbitration will continue with the remaining transfer data. In the event that two masters are arbitrating and the first master wants to put a repeated START on the bus while the second master wants to put a ZERO data bit on the bus, the first master must recognize that it cannot cause the start and lose arbitration. If the first master wants to put a repeated START on the bus while the second master wants to put a ONE data bit on the bus, the second master will see the repeated start and lose arbitration. If both masters put a repeated START on the bus in the same bit position, arbitration should continue at each data bit.

This mechanism requires that SMBus master devices participating in an arbitration cycle monitor the actual state of the SMBDAT line during arbitration.

If a master also incorporates a slave function and loses control of the bus in the arbitration cycle during the address stage, it must check the actual address placed on the bus in order to determine whether another master is trying to access it. In this case the master that lost the arbitration must switch immediately to its slave receiver mode in order to receive the rest of the message.

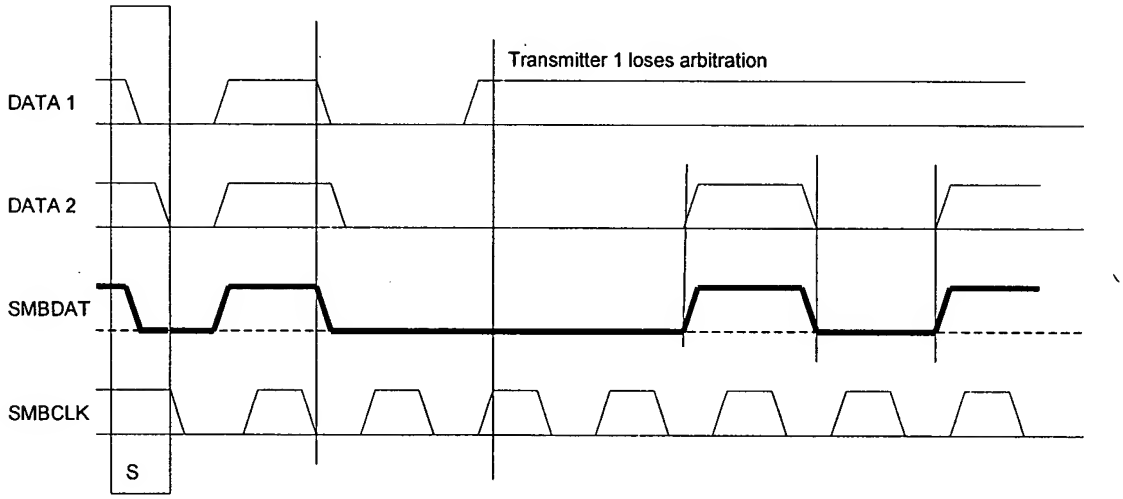


Figure 4-6: SMBus arbitration

During each bus transaction masters are still required to be able to recognize a repeated START condition on the bus. A device that detects a repeated START condition that it didn't generate must quit the transfer.

Once a master has won arbitration, arbitration is disallowed until the bus is again idle.

4.3.3. Clock low extending

SMBus provides a clock synchronization mechanism to allow devices of different speeds to co-exist on the bus. In addition to the bus arbitration procedure the clock synchronization mechanism can be used during a bit or a byte transfer in order to allow slower slave devices to cope with faster masters.

At the bit level, a device may slow down the bus by periodically extending the clock low interval.

Devices are allowed to stretch the clock during the transfer of one message up to the maximum limits described in the AC specifications of this document. Nevertheless, devices designed to stretch every clock cycle periodically must maintain the $f_{SMB,MIN}$ frequency of 10 KHz ($f_{SMB,MIN}^{-1}$= 100µs) in order to preserve the SMBus bandwidth.

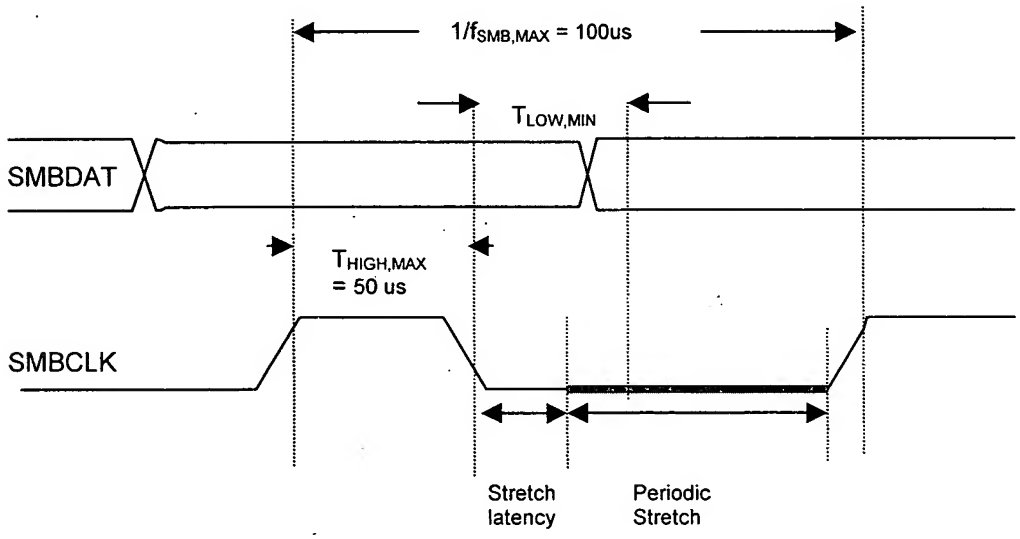


Figure 4-7: Periodic clock stretching by a slave SMBus device

Clock LOW extension, or stretching , if necessary, must start after the SMBCLK high-to-low transition within a $T_{LOW:MIN}$ - TSU:DAT interval. Devices designed to stretch the clock on every bit transfer must maintain the minimum bus frequency $f_{SMB,MIN}$ of 10 KHz. A slave device may opt to stretch the clock line during a specific bit transfer in order to process a real time task or check the validity of a byte. In this case the slave device must adhere to the $T_{TIMEOUT}$ and $T_{LOW:SEXT}$ specifications. Clock LOW extension may occur during any bit transfer including the clock provided prior to the ACK clock pulse.

A slave device may select to stretch the clock LOW period between byte transfers on the bus, in order to process received data or prepare data for transmission. In this case the slave device will hold the clock line LOW after the reception and acknowledgement of a byte. Again the slave device is responsible for not violating the $T_{LOW:SEXT}$ specification of SMBus.

During a bus transaction the master also can select to extend the clock LOW period between bytes or at any point in the byte transfer, including the clock LOW period after the byte transfer and before the acknowledgement clock. The master may need to extend the clock LOW period selectively in order to process data or serve a real time task. In doing so, the master must not exceed the $T_{LOW:MEXT}$ specification.

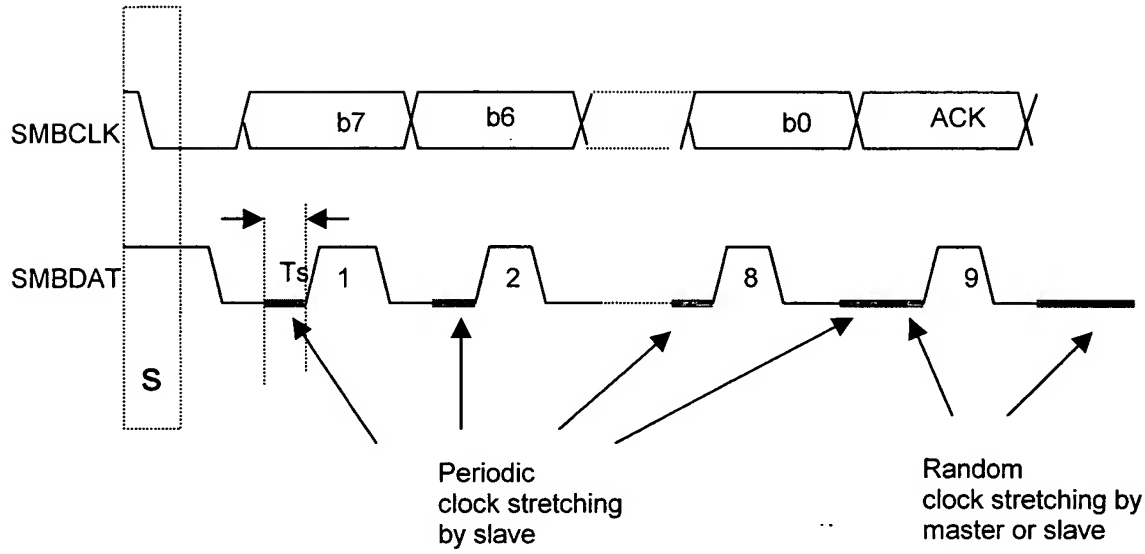


Figure 4-8: Periodic and random clock stretching

Both master and slave devices must adhere to the SMBus $T_{TIMEOUT}$ specification in order to maintain bus bandwidth and recovery from fatal bus conditions.

4.4. Data transfer formats

SMBus data transfers follow the format shown in the following figure.

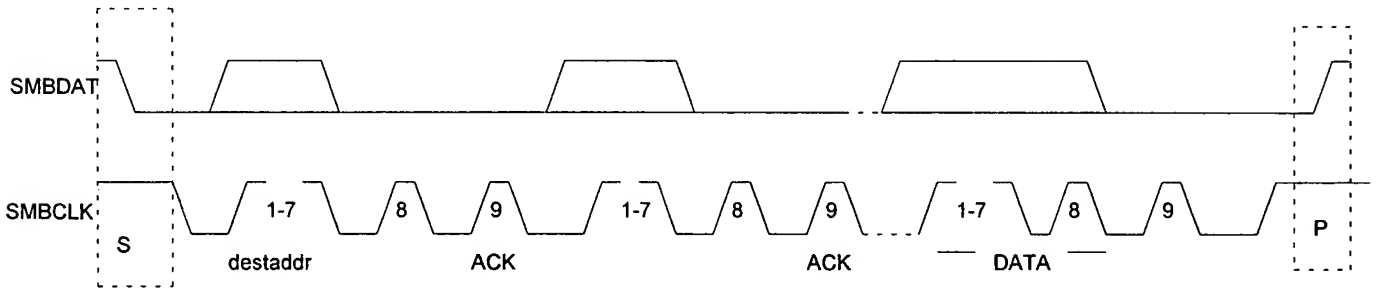


Figure 4-9: Data transfer over SMBus

After the START condition, S, the master places the 7-bit address of the slave device it wants to address on the bus. The address is 7 bits long followed by an eighth bit indicating the direction of the data transfer (R/W#); a ZERO indicates a transmission (WRITE) while a ONE indicates a request for data (READ). A data transfer is always terminated by a STOP (P) condition generated by the master.

Specific SMBus protocols require the master to generate a repeated START condition followed by the slave device address without first generating a STOP condition.

5. Layer 3 – Network layer

5.1. Usage model

The System Management Bus Specification refers to three types of devices. A *slave* is a device that is receiving or responding to a command. A *master* is a device that issues commands, generates the clocks, and terminates the transfer. A *host* is a specialized master that provides the main interface to the system's CPU. A host must be a master-slave and must support the SMBus host notify protocol. There may be at most one host in a system. One example of a hostless system is a simple battery charging station. The station might sit plugged into a wall waiting to charge a smart battery.

A device may be designed so that it is never a master and always a slave. A device may act as a slave most of the time, but in special instances it may become a master. A device can also be designed to be a master only. A system host is an example of a device that acts as a host most of the time but that includes some slave behavior..

5.2. Device identification – slave address

Any device that exists on the System Management Bus as a slave has a unique address called the *Slave Address*. For reference, the following addresses are reserved and must not be used by or assigned to any SMBus slave device unless otherwise detailed by this specification.

Slave Address Bits 7-1	R/W# bit Bit 0	Comment
0000 000	0	General Call Address
0000 000	1	START byte
0000 001	X	CBUS address
0000 010	X	Address reserved for different bus format
0000 011	X	Reserved for future use
0000 1XX	X	Reserved for future use
0101 000	X	Reserved for ACCESS.bus host
0110 111	X	Reserved for ACCESS.bus default address
1111 0XX	X	10-bit slave addressing
1111 1XX	X	Reserved for future use
0001 000	X	SMBus Host
0001 100	X	SMBus Alert Response Address
1100 001	X	SMBus Device Default Address

Table 4: Reserved SMBus Addresses

All other addresses are available for address assignment for dynamic address devices and/or for miscellaneous devices. Miscellaneous device addresses are discussed in Section 5.2.1.4.

The SMBus Alert Response Address (0001 100) can be a substitute for device master capability. See Appendix A for details.

The SMBus Device Default Address is reserved for use by the SMBus Address Resolution Protocol, which allows addresses to be assigned dynamically. See section 5.6 for details

Addresses not specified here or within the appendices are reserved for future use. All 10-bit slave addresses are reserved for future use and are outside the scope of this specification.

The host has the lowest legitimate address so that messages going to the host have the highest priority with respect to bus arbitration.

5.2.1. SMBus address types

Several SMBus devices can be used simultaneously in an actual system. In case of device address contention the designer may use either programmable features implemented in SMBus devices to resolve such contention or/and multiple SMBus branches within the same system to spread devices that use the same address.

There are several type of addresses currently in use in actual SMBus systems.

5.2.1.1. Reserved addresses

SMBus, Access.bus and I^2C reserve several addresses for specific bus functions as defined in Table 4.

5.2.1.2. Purpose-assigned addresses

These addresses are assigned by the SMBus Working Group to specific types of devices. Each device type that obtains an assigned address has to have an SMBus specification associated with it. Some systems using SMBus assume that if a device exists at a purpose-assigned address then the device complies with the associated specifications for that address. For example, SMBus application to Smart Battery implementations assume that Smart Battery devices and controllers are at their purpose-assigned addresses. Thus, devices that do not meet the purpose-assigned address specifications for Smart Battery devices cannot be used in Smart Battery applications.

Other SMBus implementations do not rely solely on the device address to identify a device's functionality. In these applications, devices may have addresses that overlap with the purpose-assigned addresses.

The device manufacturer is forewarned that this may preclude use of that device in other applications. In general, purpose-assigned addresses should be avoided except for devices that are intended to meet the specification for the corresponding address(es). The device manufacturer should consult the SMBus WG to get the latest information on purpose-assigned addresses as a guide to whether their address assignment is disallowed in certain SMBus applications.

5.2.1.3. Prototype Addresses

Slave Address	Description
1001 0XX	Prototype Addresses

Table 5: Prototype addresses

The Prototype addresses (1001 0XX) are reserved for device prototyping and experimenting in applications that utilize purpose-assigned addresses. They are not intended for production parts and should never be assigned to any device.

5.2.1.4. Miscellaneous device addresses

Manufacturers have produced and may continue producing SMBus compatible devices for specific system purposes, for which they do not need to implement the complete SMBus specification, or for which they do not require explicit support from the OS. Such devices, for example, may be port expanders, D/A circuits, etc. The SMBus web site (http://www.smbus.org) includes a page where SBS-IF member companies can share address information for such devices. This page may aid manufacturers to avoid address conflicts with devices likely to co-exist on the same SMBus segment.

5.2.1.5. Dynamically assigned addresses

Version 2.0 introduces the concept of dynamically assigned addresses. This is detailed in section 5.6.

5.3. Using a device

A typical SMBus device will have a set of commands by which data can be read and written. All commands are 8 bits (1 byte) long. Command arguments and return values can vary in length. Accessing a command that does not exist or is not supported may cause an error condition. In accordance with this specification, the Most Significant Bit is transferred first.

There are eleven possible command protocols for any given device. A slave device may use any or all of the eleven protocols to communicate. The protocols are Quick Command, Send Byte, Receive Byte, Write Byte, Write Word, Read Byte, Read Word, Process Call, Block Read, Block Write and Block Write-Block Read Process Call.

5.4. Packet error checking

Version 1.1 of SMBus introduced a Packet Error Checking mechanism to improve reliability and communication robustness. Implementation of Packet Error Checking by SMBus devices is optional for SMBus devices but is required for devices participating in and only during the ARP process. SMBus devices that implement Packet Error Checking must be capable to communicate with the host and other devices that do not implement the Packet Error Checking mechanism.

Packet Error Checking, whenever applicable, is implemented by appending a Packet Error Code (PEC) at the end of each message transfer. Each protocol (except for Quick Commandand the host notify protocol described in a later Section) has two variants: one with the Packet Error Code (PEC) byte and one without. The PEC is a CRC-8 error-checking byte, calculated on all the message bytes (including addresses and read/write bits). The PEC is appended to the message by the device that supplied the last data byte.

5.4.1. Packet error checking implementation

The SMBus must accommodate any mixture of devices that support Packet Error Checking and devices that do not. A device that acts as a slave and supports the PEC must always be prepared to perform the slave transfer with or without a PEC, verify the correctness of the PEC if present, and only process the message if the PEC is correct. Implementations are encouraged to issue a NACK if the PEC is present but not correct.

5.4.1.1. ACK/NACK and Packet Error Checking

The ACK/NACK bit in an SMBus byte is as susceptible to corruption as any other bit in an SMBus packet. Hence, an ACK at the end of a PEC is not a guarantee that the PEC is correct. A master-transmitter receiving an ACK at the end of a write should not necessarily assume that the write was successfully carried out at the slave-receiver of the write, although it is highly likely that it was.

A NACK received after a PEC by a master-transmitter indicates that the link layer of the slave-receiver became aware of an error with the transmission in time to supply a NACK at the end of the PEC byte. This may be due to an incorrect PEC or any other error. Errors discovered above the data link layer may also be indicated with a NACK if the device is fast enough to discover and indicate the error when the NACK is due.

An ACK received after a PEC by a master-transmitter means that no error could be determined by the link layer in the slave-receiver in time to supply a NACK. This might be because the receiver is not able to check the validity of the PEC in real time.

If a master transmitter wants to be sure that a write-operation is performed correctly at the target device, it must use some higher-layer mechanism to verify this. This might take the form of a read-with-PEC of the data; receipt of a correct PEC would reliably indicate that that the original write occurred without error.

When a master-receiver reads data from a slave-transmitter, the ACK/NACK supplied by the master-receiver at the end of the transaction has little meaning other than to mark the end of the last byte. The slave-transmitter is supplying the data, and if the PEC supplied by the slave-transmitter is correct, the master-receiver may assume that the data was received as the slave transmitted it. If not, it is up to the master-receiver to take any appropriate remedial action.

5.4.1.2. Master implementation

A master may use PEC on any transaction. It is required that the master have either a priori knowledge of whether or not the target slave supports PEC or a way to determine whether the target slave supports PEC. The use of PEC is governed by upper layer protocols (e.g. device drivers), specifications (e.g.requirements of the SMBus ARP protocol) or detection algorithms for a given class of devices (e.g. smart batteries).

5.4.1.3. Slave implementation

A slave device that implements Packet Error Checking must be prepared to receive and transmit data with or without a PEC. During a slave receive transfer, after the device has identified the protocol and command must accept and check the additional PEC for validity.

During a slave transmit transfer, the slave transmitter must respond to additional clocks after the last byte transfer and furnish a PEC to the master receiver requesting it.

Each bus transaction requires a Packet Error Code (PEC) calculation by both the transmitter and receiver within each packet. The PEC uses an 8-bit cyclic redundancy check (CRC-8) of each read or write bus transaction to calculate a Packet Error Code (PEC). The PEC may be calculated in any way that conforms to a CRC-8 represented by the polynomial, $C(x) = x^8 + x^2 + x^1 + 1$ and must be calculated in the order of the bits as received.

Calculating the PEC for transmission or reception is implemented in a method chosen by the device manufacturer. It is possible to perform the check with low-cost hardware or firmware algorithm that could process the message bit-by-bit or with a byte-wise look-up table. The SMBus web page provides some example CRC-8 methods.

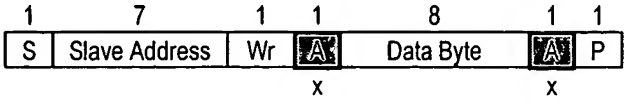
The PEC is appended to the message as dictated by the protocols in section 5.5. The PEC calculation includes all bytes in the transmission, including address, command and data. The PEC calculation does not include ACK, NACK, START, STOP nor Repeated START bits. This means that the PEC is computed over the entire message from the first START condition.

Whether a device implements packet error checking may be determined by the specification revision code that is present in the SpecificationInfo() command for a Smart Battery, Smart Battery Charger or Smart Battery Selector. See these individual specifications for exact revision coding identities. It may also be discovered in the UDID, defined in section 5.6.1, for other devices.

5.5. Bus Protocols

Following is a description of the various SMBus protocols with and without a Packet Error Code. Compliant devices need not support all the protocols defined in this section. The results returned by such a device to a protocol it does not support are undefined.

Below is a key to the protocol diagrams in this section. Not all protocol elements will be present in every command. For instance, not all packets are required to include the Packet Error Code.



S	Start Condition
Sr	Repeated Start Condition
Rd	Read (bit value of 1)
Wr	Write (bit value of 0)
x	Shown under a field indicates that that field is required to have the value of 'x'
A	Acknowledge (this bit position may be '0' for an ACK or '1' for a NACK)
P	Stop Condition
PEC	Packet Error Code
☐	Master-to-Slave
■	Slave-to-Master
...	Continuation of protocol

Figure 5-1: SMBus packet protocol diagram element key

A value shown below a field in the following diagrams is a mandatory value for that field.

The data formats implemented by SMBus are:

- Master-transmitter transmits to slave-receiver: The transfer direction in this case is not changed.
- Master reads slave immediately after the first byte: At the moment of the first acknowledgment (provided by the slave-receiver) the master-transmitter becomes a master-receiver and the slave-receiver becomes a slave-transmitter.
- Combined format: During a change of direction within a transfer, the master repeats both a START condition and the slave address but with the R/W# bit reversed. In this case the master receiver terminates the transfer by generating a NACK on the last byte of the transfer and a STOP condition.

Examples of these formats will be seen in the SMBus protocols below.

5.5.1. Quick command

Here, part of the slave address denotes the command – the R/W# bit. The R/W# bit may be used to simply turn a device function on or off, or enable/disable a low-power standby mode. There is no data sent or received.

The quick command implementation is good for very small devices that have limited support for the SMBus specification. It also limits data on the bus for simple devices.

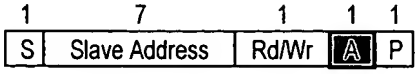


Figure 5-2: Quick command protocol

5.5.2. Send byte

A simple device may recognize its own slave address and accept up to 256 possible encoded commands in the form of a byte that follows the slave address.

All or parts of the Send Byte may contribute to the command. For example, the highest 7 bits of the command code might specify an access to a feature, while the least significant bit would tell the device to turn the feature on or off. Or, a device may set the "volume" of its output based on the value it received from the Send Byte protocol.

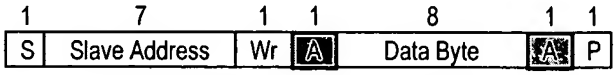


Figure 5-3: Send byte protocol

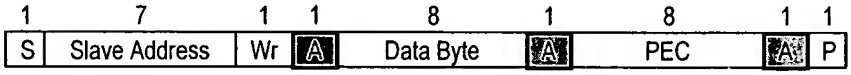


Figure 5-4: Send byte protocol with PEC

5.5.3. Receive byte

The Receive Byte is similar to a Send Byte, the only difference being the direction of data transfer. A simple device may have information that the host needs. It can do so with the Receive Byte protocol. The same device may accept both Send Byte and Receive Byte protocols. A NACK (a '1' in the ACK bit position) signifies the end of a read transfer.

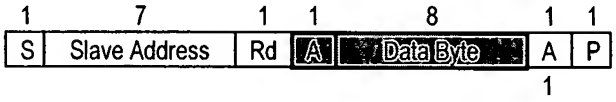


Figure 5-5: Receive byte protocol

Figure 5-6: Receive byte protocol with PEC

5.5.4. Write byte/word

The first byte of a Write Byte/Word access is the command code. The next one or two bytes, respectively, are the data to be written. In this example the master asserts the slave device address followed by the write bit. The device acknowledges and the master delivers the command code. The slave again acknowledges before the master sends the data byte or word (low byte first). The slave acknowledges each byte, and the entire transaction is finished with a STOP condition.

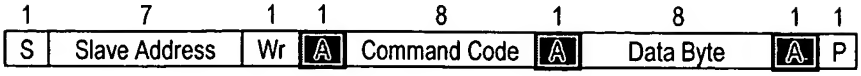


Figure 5-7: Write byte protocol

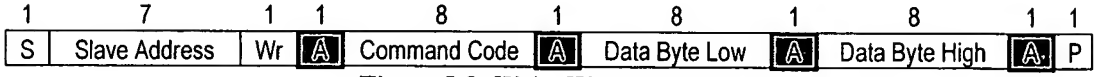


Figure 5-8: Write Word Protocol

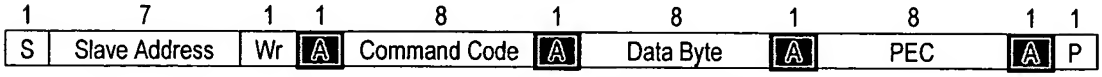


Figure 5-9: Write byte protocol with PEC

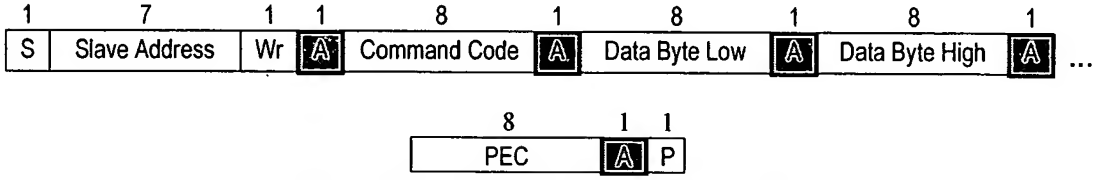


Figure 5-10: Write Word Protocol with PEC

5.5.5. Read byte/word

Reading data is slightly more complicated than writing data. First the host must write a command to the slave device. Then it must follow that command with a repeated START condition to denote a read from that device's address. The slave then returns one or two bytes of data.

Note that there is no STOP condition before the repeated START condition, and that a NACK signifies the end of the read transfer.

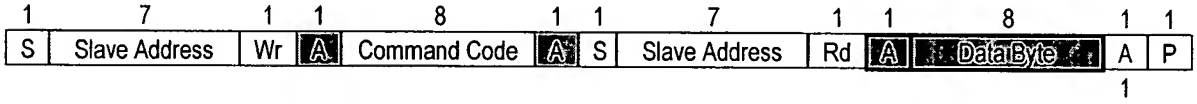


Figure 5-11: Read Byte Protocol

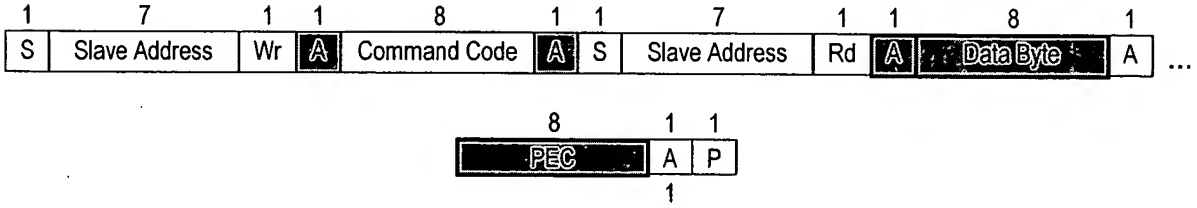


Figure 5-12: Read byte protocol with PEC

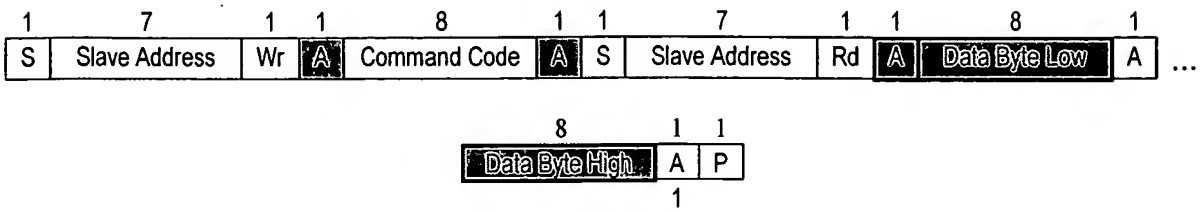


Figure 5-13: Read word protocol

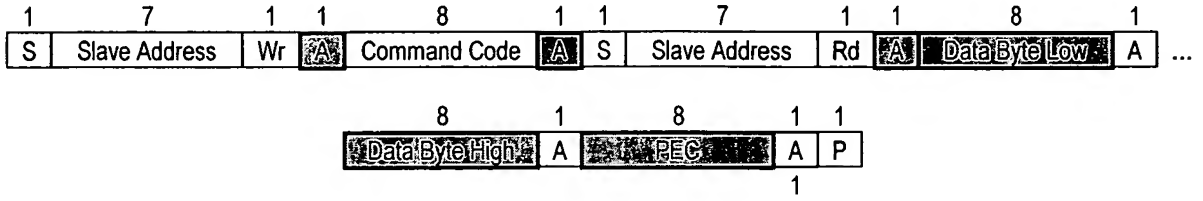


Figure 5-14: Read word protocol with PEC

5.5.6. Process call

The process call is so named because a command sends data and waits for the slave to return a value dependent on that data. The protocol is simply a Write Word followed by a Read Word without the Read-Word command field and the Write-Word STOP bit.

Note that there is no STOP condition before the repeated START condition, and that a NACK signifies the end of the read transfer.

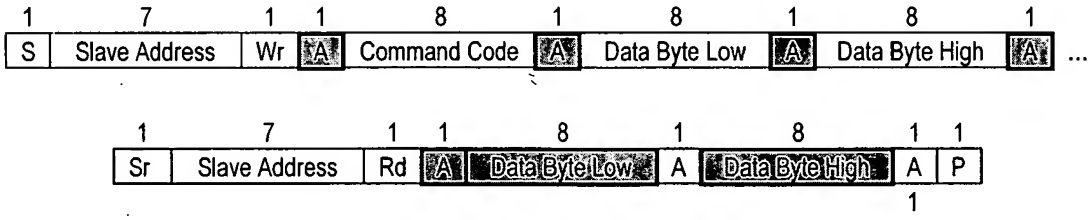


Figure 5-15: Process Call

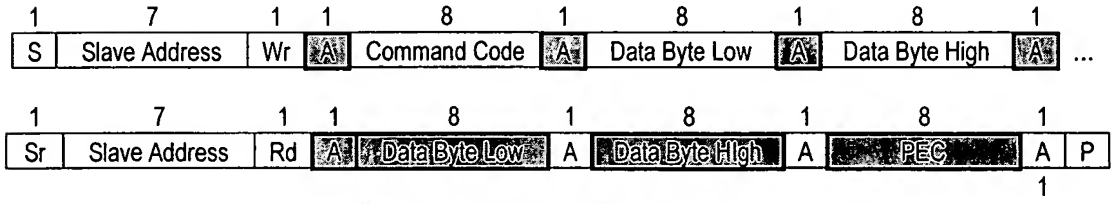


Figure 5-16: Process Call with PEC

5.5.7. Block write/read

The Block Write begins with a slave address and a write condition. After the command code the host issues a byte count which describes how many more bytes will follow in the message. If a slave has 20 bytes to send, the byte count field will have the value 20 (14h), followed by the 20 bytes of data. The byte

count does not include the PEC byte. The byte count may not be 0. A Block Read or Write is allowed to transfer a maximum of 32 data bytes.

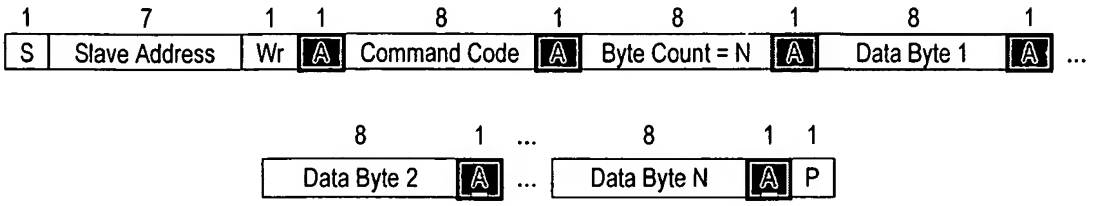


Figure 5-17: Block Write

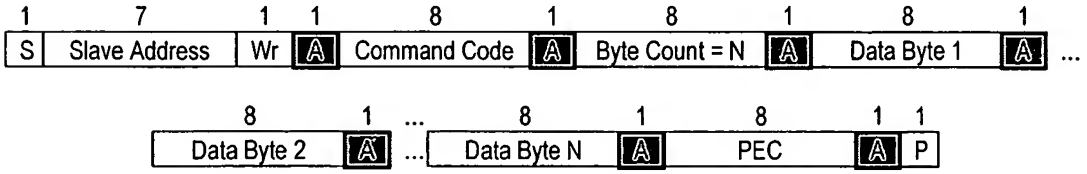


Figure 5-18: Block Write with PEC

A Block Read differs from a block write in that the repeated START condition exists to satisfy the requirement for a change in the transfer direction. A NACK immediately preceding the STOP condition signifies the end of the read transfer.

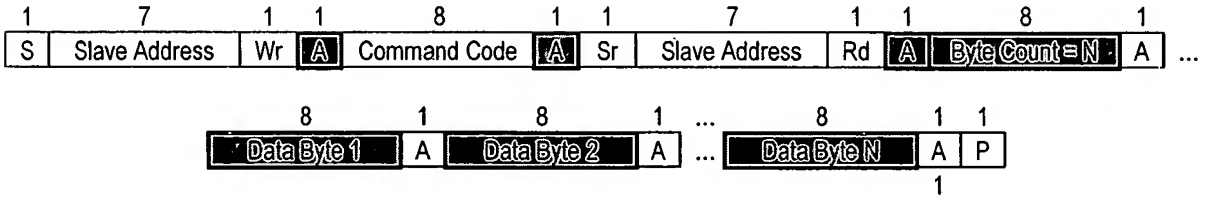


Figure 5-19: Block Read

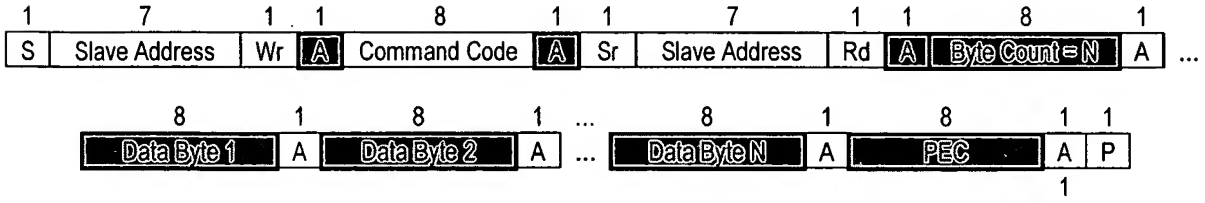


Figure 5-20: Block Read with PEC

5.5.8. Block write-block read process call

The block write-block read process call is a two-part message. The call begins with a slave address and a write condition. After the command code the host issues a write byte count (M) that describes how many more bytes will be written in the first part of the message. If a master has 6 bytes to send, the byte count

field will have the value 6 (0000 0110b), followed by the 6 bytes of data. The write byte count (M) cannot be zero.

The second part of the message is a block of read data beginning with a repeated start condition followed by the slave address and a Read bit. The next byte is the read byte count (N), which may differ from the write byte count (M). The read byte count (N) cannot be zero.

The combined data payload must not exceed 32 bytes. The byte length restrictions of this process call are summarized as follows:

- $M \geq 1$ byte
- $N \geq 1$ byte
- $M + N \leq 32$ bytes

The read byte count does not include the PEC byte. The PEC is computed on the total message beginning with the first slave address and using the normal PEC computational rules. It is highly recommended that a PEC byte be used with the Block Write-Block Read Process Call.

Note that there is no STOP condition before the repeated START condition, and that a NACK signifies the end of the read transfer.

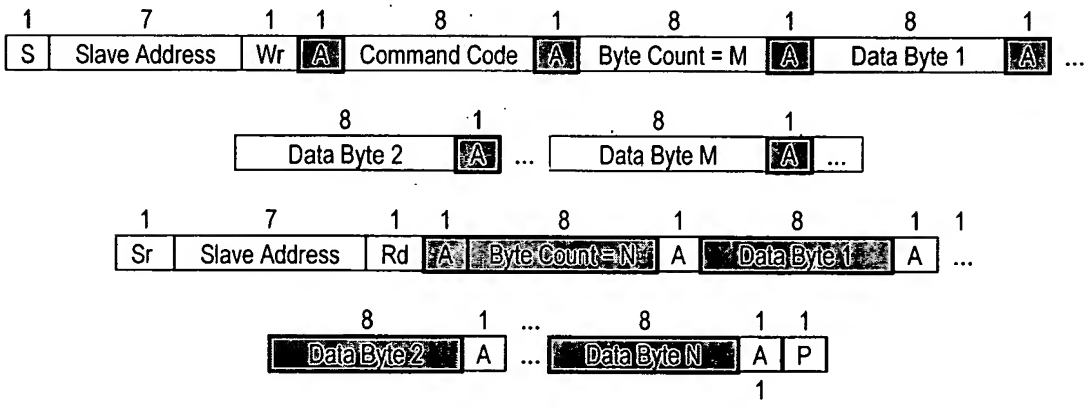


Figure 5-21: Block Write - Block Read Process Call

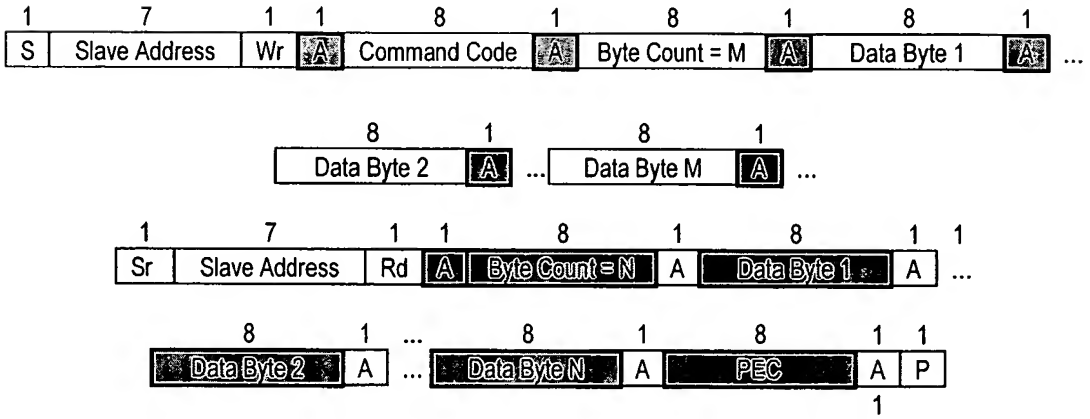


Figure 5-22: Block Write - Block Read Process Call with PEC

5.5.9 SMBus host notify protocol

To prevent messages coming to the SMBus host controller from unknown devices in unknown formats only one method of communication is allowed, a modified form of the Write Word protocol. The standard Write Word protocol is modified by replacing the command code with the alerting device's address. This protocol MUST be used when an SMBus device becomes a **master** in order to communicate with the SMBus host acting as a **slave**.

Communication from SMBus Device to SMBus Host begins with the SMBus Host address (0001 000b). The message's Command Code is the initiating SMBus device's address. From this, the SMBus Host knows the origin of the following 16 bits of device status. The contents of the status are device specific.

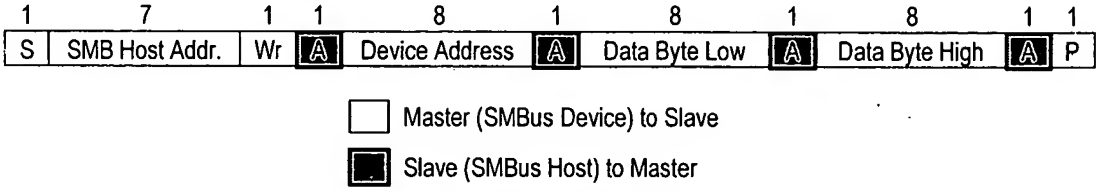


Figure 5-23: 7-bit Addressable Device to Host Communication

SMBus hosts must support the host notify protocol. Hosts may implement the optional #SMBALERT line if devices in the system use it.

5.6. SMBus Address resolution protocol

SMBus slave address conflicts can be resolved by dynamically assigning a new unique address to each slave device. The Address Resolution Protocol (ARP) possesses the following attributes:

- Address assignment utilizes the standard SMBus physical layer arbitration mechanism
- Assigned addresses remain constant while device power is applied; address retention through device power loss is also allowed
- No additional SMBus packet overhead is incurred after address assignment. (i.e. subsequent accesses to assigned slave addresses have the same overhead as accesses to fixed address devices.)
- Any SMBus master can enumerate the bus

5.6.1. Unique Device Identifier (UDID)

In order to provide a mechanism to isolate each device for the purpose of address assignment each device must implement a unique device identifier (UDID). This 128-bit number is comprised of the following fields:

8 bits	8 bits	16 bits	16 bits	16 bits	16 bits	16 bits	32 bits
Device Capabil-ities	Version / Revision	Vendor ID	Device ID	Interface	Subsystem Vendor ID	Subsystem Device ID	Vendor Specific ID

Msb Lsb

Figure 5-24: UDID

Device Capabilities Describes the device's capabilities. See detail below.

Version / Revision	UDID version number, and a silicon revision identification. See detail below.
Vendor ID	The device manufacturer's ID as assigned by the SBS Implementers' Forum or the PCI SIG.
Device ID	The device ID as assigned by the device manufacturer (identified by the Vendor ID field).
Interface	Identifies the protocol layer interfaces supported over the SMBus connection by the device. For example, ASF and IPMI.
Subsystem Vendor ID	This field may hold a value derived from any of several sources: • The device manufacturer's ID as assigned by the SBS Implementers' Forum or the PCI SIG. • The device OEM's ID as assigned by the SBS Implementers' Forum or the PCI SIG. • A value that, in combination with the Subsystem Device ID, can be used to identify an organization or industry group that has defined a particular common device interface specification.
Subsystem Device ID	The subsystem ID identifies a specific interface, implementation, or device. The Subsystem ID is defined by the party identified by the Subsystem Vendor ID field.
Vendor-specific ID	A unique number per device. See detail below.

5.6.1.1. Device capabilities field

The Device Capabilities field serves multiple purposes:

1. Reports generic SMBus capabilities.
2. Guarantees order of ARP resolution. Because a 'zero' bit wins arbitration over a '1' bit, and the Address Type bits are the first two bits presented when a device presents its UDID, all fixed address devices on the bus are detected during ARP first, followed by devices with dynamic and persistent addresses, and so on. The bits in the brackets below show the highest two bits, the address type field, within the Device Capabilities field:

- [00] Fixed Address devices are identified first.
- [01] Dynamic and Persistent Address devices are identified next.
- [10] Dynamic and Volatile Address devices are identified next.
- [11] Random Number devices are identified last.

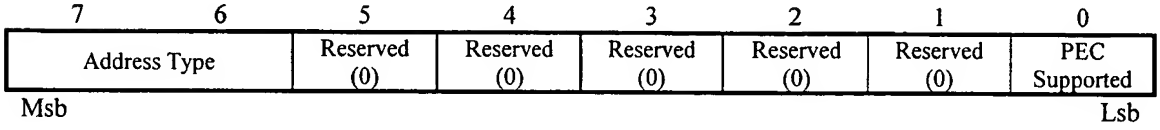


Figure 5-25: 8-bit device capabilities field

Address Type	These two bits describe the type of address contained in the device: 00 Fixed Address device 01 Dynamic and persistent address device 10 Dynamic and volatile address device 11 Random number device
PEC Supported	This bit is set if the device supports Packet Error Code on all commands supported at the device's SMBus address associated with this UDID. If this bit is not set, the

ability of the device to support PEC is unknown.

Reserved Reserved bits are for future extendibility and must be returned as 0b and ignored when read.

5.6.1.2. Version/Revision field

The version/revision field serves multiple purposes:

1. Identifies a UDID version to allow for future extendibility.
2. Identifies the silicon revision level.

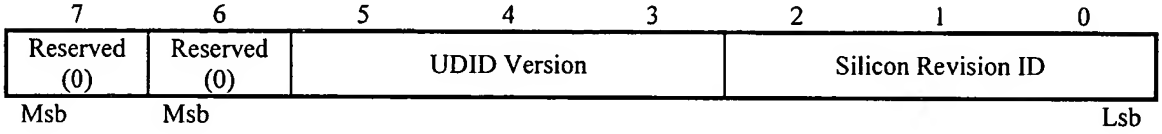


Figure 5-26: Version/Revision field

Reserved Reserved bits are for future extendibility and must be a 0b.

UDID Version These bits define the UDID version as defined here:

- 000b Reserved
- 001b UDID version 1 (defined for SMBus 2.0 release)
- 010b – 111b Reserved for future use

It is expected that the version will increment as the bit definitions or protocols in this section change

Silicon Revision ID These bits are used to designate the silicon revision level. The vendor determines this value. The vendor is encouraged to increment this value when silicon changes are made that will/might affect the software interface (e.g. new features, changed interface, etc.). In the event that all 8 encoded values are exhausted, the vendor is encouraged to use a different Device ID for the next revision.

5.6.1.3. Interface

The Interface field defines the SMBus version and the Interface Protocols supported.

The least significant nibble is used to identify the SMBus version:

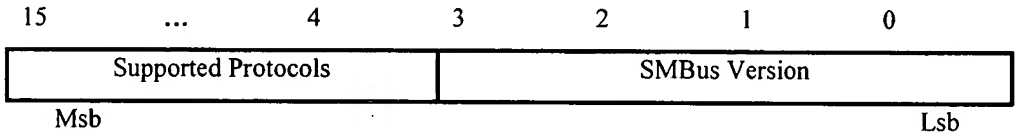


Figure 5-27: Interface field

SMBus Version These bits define the SMBus version as defined here:

- 0000b SMBus 1.0 – do not use in ARPable devices
- 0001b SMBus 1.1– do not use in ARPable devices

- 0010b Reserved
- 0011b Reserved
- 0100b SMBus Version 2.0
- All other values reserved

Note: The values 0000b and 0001b in the field definition above support use of the UDID definition in other specifications.

The most significant bits of the interface field are used to identify the protocols supported by the device:

Protocols	Meaning
bit 15	Reserved for future definition under the SMBus specifications.
bit 14	Reserved for future definition under the SMBus specifications.
bit 13	Reserved for future definition under the SMBus specifications.
bit 12	Reserved for future definition under the SMBus specifications.
bit 11	Reserved for future definition under the SMBus specifications.
bit 10	Reserved for future definition under the SMBus specifications.
bit 9	Reserved for future definition under the SMBus specifications.
bit 8	Reserved for future definition under the SMBus specifications.
bit 7	Reserved for future definition under the SMBus specifications.
bit 6 IPMI	Device supports additional interface access and capabilities per IPMI specifications
bit 5 ASF	Device supports additional interface access and capabilities per ASF specifications
bit 4 OEM	Device supports vendor-specific access and capabilities per the Subsystem Vendor ID and Subsystem Device ID fields returned by discoverable SMBus devices. The Subsystem Vendor ID identifies the vendor or defining body that has specified the behavior of the device. The Subsystem Device ID is used in conjunction with the System Vendor ID to specify a particular level of functional equivalence for the device.

5.6.1.4. SubSystem IDs

The SubSystem Vendor ID can be specified as 0000h if the SubSystem fields are unsupported. If the SubSystem Vendor ID is 0000h, the SubSystem Device ID must also be 0000h. These fields may not be supported for inexpensive or generic type sensors that do not require subsystem identification/differentiation. If these fields are supported, it is required that the values be stored in some form of non-volatile storage.

5.6.1.5. Vendor-specific ID

This field is used to provide a unique ID for functionally equivalent devices. This is for devices that would otherwise return identical UDIDs for the purpose of address assignment. This field is defined by the device manufacturer (as specified by the Vendor ID field) who may employ a central numbering scheme or a random number scheme for dynamic address devices. The data in this field is irrelevant for devices that do not support dynamic addressing.

The rules of this field are stated here for clarity:

1. Devices that support an assigned device address must support a unique ID in this field.

2. If a pre-assigned unique ID is used, at least 16-bits must be unique. However, 32-bits is recommended.
3. If a random number is implemented in this field, Random Number Requirements must be met.
4. Devices that support a fixed device address must still implement this field but not uniquely.

Uniqueness is important to guarantee that two like devices are identified discretely. It is the responsibility of the device/system manufacturer to determine the possibility of like devices, and the mechanism for providing uniqueness via the UDID and slave address fields.

5.6.1.6. Random number requirements

If a random number is utilized the following requirements must be met:

1. It must be at least 16 bits in length.
2. The device is not allowed to support a persistent slave address.
3. The device is not allowed to support fixed addresses. (If the device has a fixed address mode, the Vendor Specific ID should be a constant, and therefore not random – this is required to guarantee that the SMBus ARP resolution order is maintained)
4. The random number must be retained while the device has power with the exceptions described in items 5 and 6.
5. The random number must be regenerated when the device receives the Reset Device command.
6. The random number must be regenerated when the device senses a bus collision during a read operation directed to its Assigned Slave Address. When this happens, the device must issue a host notify protocol if the device supports it.

5.6.2. Power-on reset

Power-on reset is described in section 3.1.4.2. In the case of ARP-capable devices, 'operational state' implies the ability to respond to ARP commands as required in this section.

Each slave device must fit into only one of these categories and must obey the power on reset state:

Device Type	AR Flag	AV Flag	SMB Address	UDID
PSA (Persistent Slave Address)	CLEAR	Read from NVR	Read from NVR; undefined if AV Flag is CLEAR	NO CHANGE
Non-PSA / Non-Random Number	CLEAR	CLEAR	undefined	NO CHANGE
Non-PSA / Random Number	CLEAR	CLEAR	undefined	Generate Random Number

Table 6: Internal state of ARP-capable devices on power-on reset

5.6.3. ARP commands

The ARP Master can issue general commands and directed commands. A general command is targeted at all devices and is required for the address resolution process. A directed command is targeted at a single device. All packets originated by the ARP Master use Packet Error Checking (See Section 7.5) and begin with the basic format:

<SMBus Device Default Address> <command>

<SMBus Device Default Address>	1100 001 (the R/W# bit completes the byte)
<command>	*General* (0x00 through 0x1F) 0x00 = Reserved 0x01 = Prepare to ARP 0x02 = Reset Device (general) 0x03 = Get UDID (general) 0x04 = Assign Address 0x05 = Reserved • • • 0x1F = Reserved *Directed* • Odd numbered commands denote Get UDID from a specific slave and are of the form yyyy yyy1 where yyyy yyy is the 7-bit targeted slave address. For example, a command of 0x21 is a directed Get UDID to slave address 0010 000. • Even numbered commands denote Reset Device to a specific slave and are of the form yyyy yyy0 where yyyy yyy is the 7-bit targeted slave address. For example, a command of 0x5C is a directed Reset Device to slave address 0101 110. • Values of 0xFE and 0xFF are reserved

Table 7: ARP command number scheme

An ARP Enumerator is allowed to issue the "Prepare to ARP", "Get UDID" (general and directed), and "Assign Address" commands; it is not allowed to issue the Reset Device commands. It must execute the "Assign Address" command for each device in the general "Get UDID" command using the same address that is returned by the "Get UDID" command. An SMBus Enumerator is not allowed to re-assign addresses and it is not allowed to assign an address to a device with an invalid/uninitialized address.

Devices can optionally support the "Notify ARP Master" command that is used to notify the ARP Master that the device requires address resolution. If the ARP Master supports this command, it must respond as a slave to this command and provide a software indication that the ARP needs to be executed.

5.6.3.1. Device categorization

SMBus devices are categorized as follows:

ARP-capable	Device supports all ARP commands with the exception of the optional host notify command. Slave address is assignable. Device supports both Reset commands.
Fixed and Discoverable	Device supports the Prepare to ARP, directed Get UDID, general Get UDID and Assign Address commands. Slave address is fixed; device will accept the Assign Address command but will not allow address reassignment. Device supports both Reset commands.
Fixed, not Discoverable	Device supports the directed Get UDID command. Slave address is fixed.
Non-ARP-capable	Device does not support any ARP commands. Slave address is fixed.

5.6.3.2. Prepare to ARP

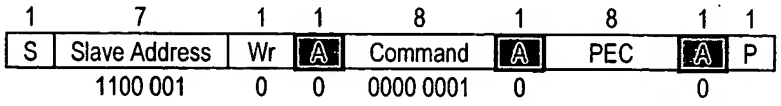
Action: always ACK/PROCESS

AR Flag: CLEAR

AV Flag: NO CHANGE

This command informs all devices that the ARP Master is starting the ARP process. All ARP-capable devices must acknowledge all bytes in this SMBus packet and clear their Address Resolved (AR) flag. They must also cancel any pending "Notify ARP Master" commands. If the ARP Master detects that any of the bytes have not been acknowledged then it can assume that there are no ARP-capable devices present on the bus. Retries are recommended in case bus noise causes an erroneous NACK.

This command utilizes the standard SMBus Send Byte Protocol with PEC as illustrated below.



5.6.3.3. *Reset device (general)*

Action: always ACK/PROCESS

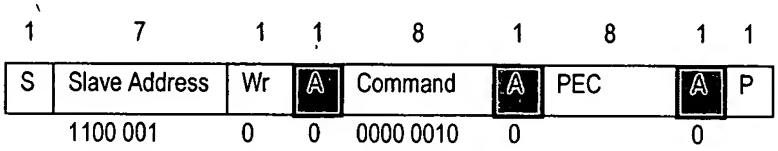
AR Flag: CLEAR

AV Flag: if (non-PSA) then CLEAR; else NO CHANGE

This command forces all non-PSA, ARP-capable devices to return to their initial state. That is, they must clear their AR (Address Resolved) and AV (Address Valid) flags; those devices that support the Persistent Slave Address must clear their AR flag. An ARP-capable device that implements a random number as part of its UDID must regenerate its random number upon receipt of this command. All ARP-capable devices must acknowledge all bytes in this SMBus packet. If the ARP Master detects that any of the bytes have not been acknowledged then it can assume that there are no ARP-capable devices present on the bus.

This reset is just for the ARP functions, and is not intended as a general device reset.

This command utilizes the standard SMBus Send Byte Protocol with PEC as illustrated below.

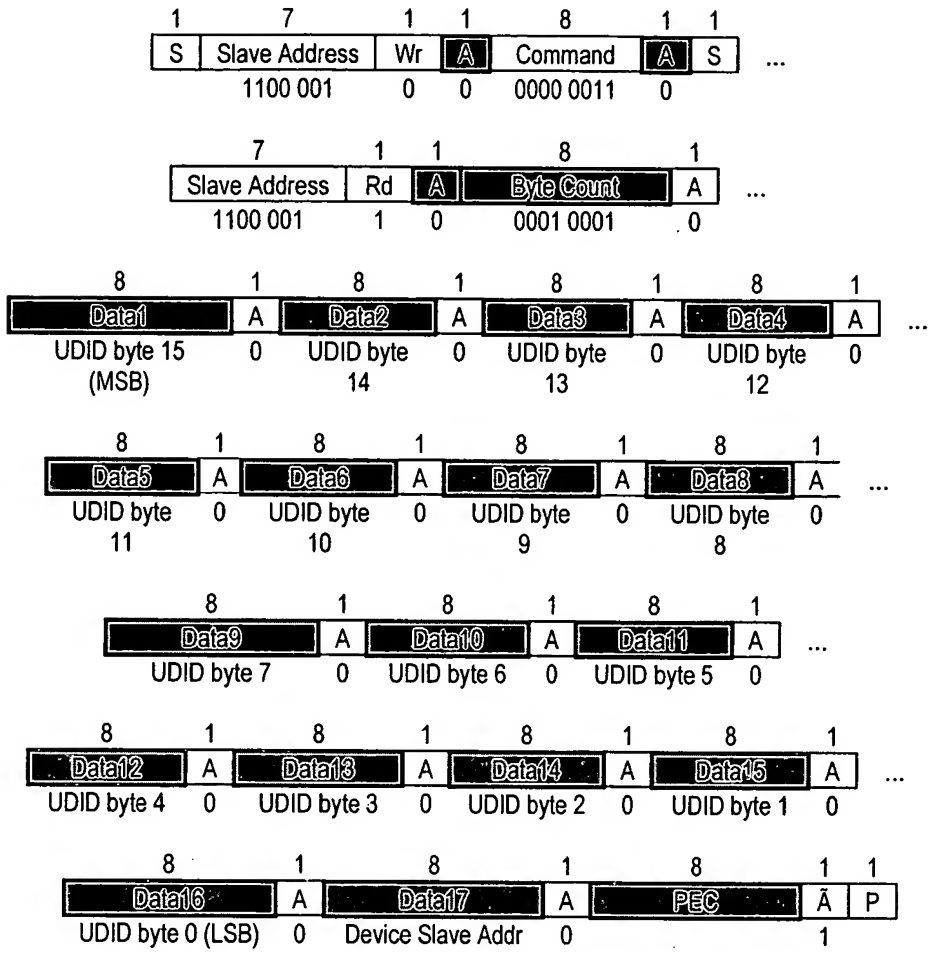


5.6.3.4. *Get UDID (general)*

Action: if (AR=0) then ACK/PROCESS; else NACK/REJECT.
AR Flag: NO CHANGE
AV Flag: NO CHANGE

This command requests ARP-capable and/or Discoverable devices to return their slave address along with their UDID. If the ARP Master detects that any of the first three bytes have not been acknowledged then it can assume that there are no ARP-capable or Discoverable devices present on the bus or all ARP-capable devices have valid assigned slave addresses.

This command utilizes the standard SMBus Block Read Protocol with PEC as illustrated below.



NOTES

- Bit 0 (lsb) in the Data17 field must be returned as 1.
- If a device has its AV flag clear then it must return 1111 111 for the remaining bits in the Data17 field.

5.6.3.5. Assign address

Action: always ACK; if (UDID match) then PROCESS.

AR Flag: SET if UDID matches.

AV Flag: SET if UDID matches.

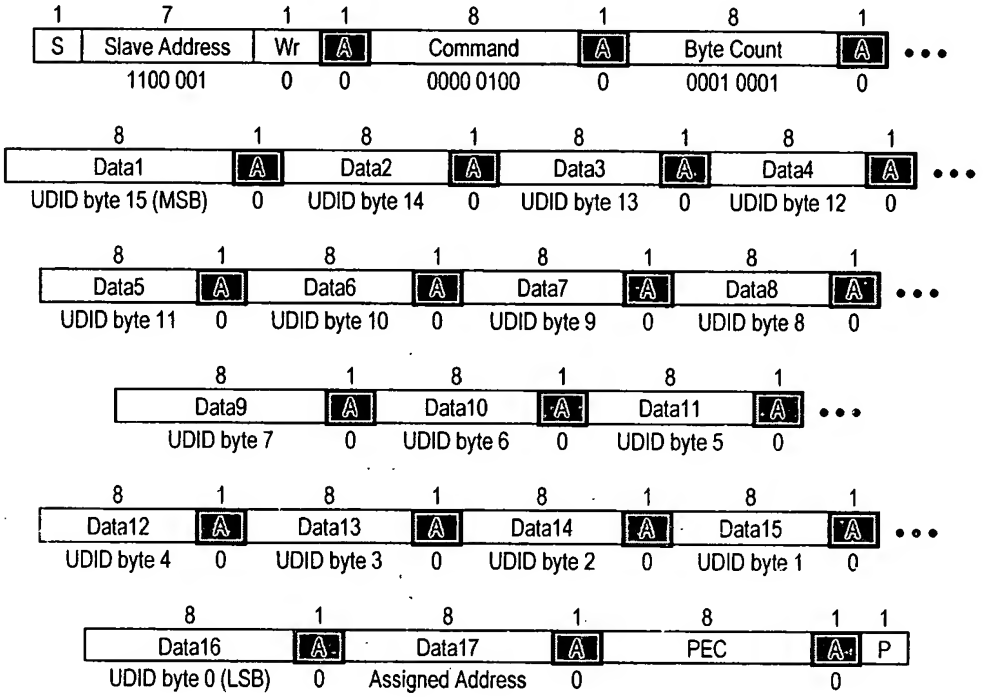
The ARP Master assigns an address to a specific device with this command. Since this command utilizes a particular device's UDID only that device will adopt the new address. All ARP-capable devices must monitor the UDID bytes in this packet (all bytes except the assigned address byte) . Once a device determines that it is not the target of the command (due to a UDID bit or byte mismatch) it must NACK the current byte, if possible, or the next byte. A slave device matching all but the last UDID byte has the choice to NACK the last UDID byte or the subsequent assigned address byte. If the ARP Master detects a NACK for any byte then it must assume that the target device is no longer present. It is suggested that the ARP Master consider retrying the command in order to guard against noise causing a present device to be ignored.

A slave device that matches the entire UDID must immediately adopt the new slave address. It must reprogram its Persistent Slave Address, if applicable. Bit 0 (lsb) of the Assigned Address field must be ignored.

NOTES:

1. A slave device must respond to this command even if its AR flag is SET.
2. The slave device only ACKs the PEC byte if it matches the value calculated on data it received, if not it must NACK the PEC byte AND ignore the "Assign Address" command. This behavior allows the host to determine that the slave device successfully accepted the address without any further bus activity.

This command utilizes the standard SMBus Block Write Protocol with PEC as illustrated below.



5.6.3.6. *Get UDID (directed)*

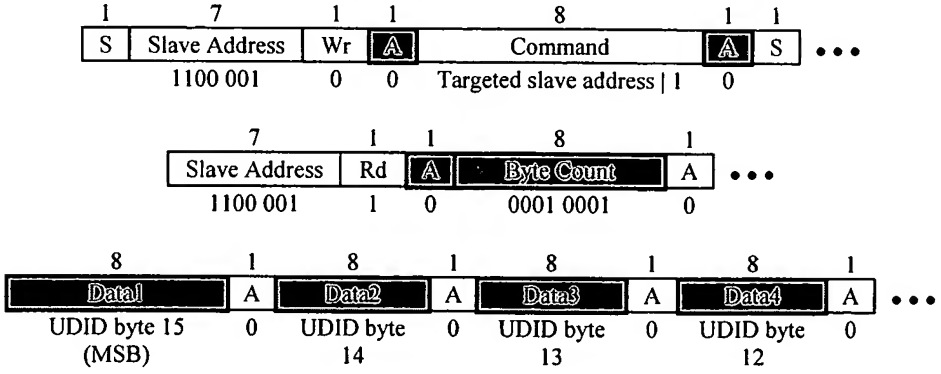
Action: if (AV=1) then ACK/PROCESS; else NACK/REJECT.

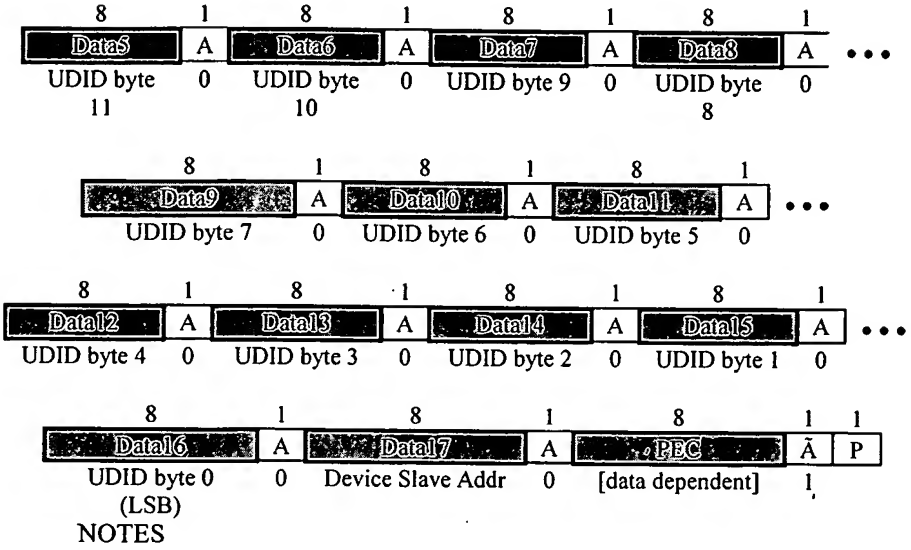
AR Flag: NO CHANGE

AV Flag: NO CHANGE

This command requests a specific ARP-capable device to return its Unique Identifier. If the ARP Master detects that any of the first three bytes have not been acknowledged then it can assume that no ARP-capable device is present at the targeted slave address.

This command utilizes the standard SMBus Block Read Protocol with PEC as illustrated below.





NOTES

Bit 0 (lsb) in the Data17 field must be returned as 1.

5.6.3.7. *Reset device ARP (directed)*

Action: if (AV=1) then ACK/PROCESS; else NACK/REJECT.

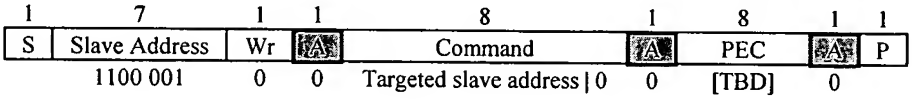
AR Flag: CLEAR

AV Flag: if (non-PSA) then CLEAR; else NO CHANGE

This command forces a specific non-PSA, ARP-capable device to return to its initial state. That is, it must clear its AR and AV flags; if the device supports the Persistent Slave Address it must clear its AR flag. An ARP-capable device that implements a random number as part of its UDID must regenerate its random number upon receipt of this command. If the ARP Master detects that any of the bytes have not been acknowledged then it can assume that no ARP-capable device is present at the targeted slave address.

This reset is just for the ARP functions, and is not intended as a general device reset.

This command utilizes the standard SMBus Send Byte Protocol with PEC as illustrated below.

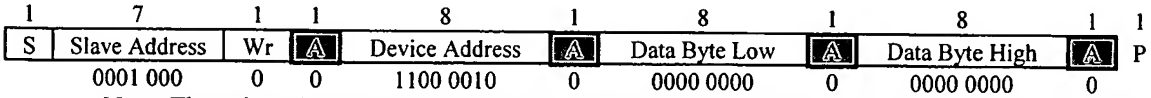


5.6.3.8. *Notify ARP master*

A device may use this command to notify the ARP Master that the device requires address resolution. The device may execute this command under the following circumstances:

- Device power up
- When the device senses a bus collision during a read operation directed to its Assigned Slave Address.

This command utilizes the standard SMBus protocol to communicate with the Host (at the SMBus Host Address as defined in 7.3.1) as illustrated below.



Note: The value of 0x0000 in the data field means that the device wishes to be ARPed. All other values are reserved for future use.

5.6.3.9. Implementation notes

An SMBus ARP Master in a Hot Plug System will typically not require the host notify command as it gets asynchronous indication of a device added or removed via other means, though there's no restriction in this specification against using this notification in those types of applications.

A mobile system that has addition and removal of devices on the SMBus can benefit from this command if the SMBus ARP Master and the removable devices support this command. (Note, there could be one device in the system that performs the notify on behalf of other devices. System software must always run the FULL ARP process.)

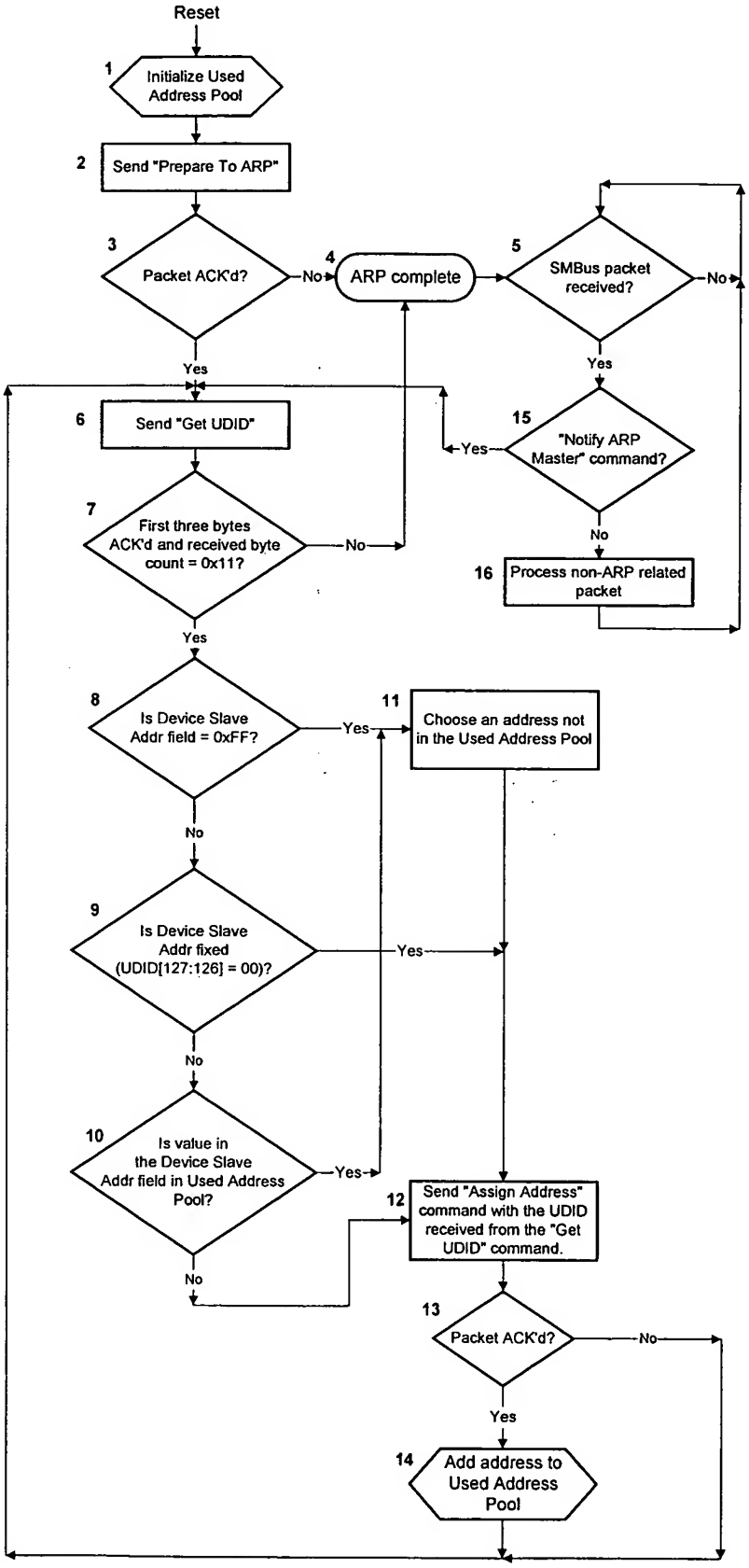
5.6.3.10. ARP execution

The ARP Master must always execute the ARP when it enters the working state and anytime it receives an SMBus status change indication (device added or removed – e.g. hot plug). The process begins with the ARP Master issuing the "Prepare To ARP" command. In all cases the ARP Master must be able to resolve addresses when it receives the "Notify ARP Master" command.

If the possibility exists that SMBus devices may join the system without a corresponding system reset (for example in hot-plug-capable systems), the ARP Master may optionally choose to issue the Get UDID (General) command at least once every 10 seconds in order to discover newly added devices that require address resolution but that don't support the "Notify ARP Master" command. No device whose AR flag is clear will respond to this command. However, a newly added device will enter the system with a power-on reset, which will reset its AR flag. It will respond to a Get UDID (General) command with its UDID. The host may choose to assign such a newly added device a non-conflicting address or it may choose to re-ARP the entire bus.

The ARP Master or any other SMBus master can perform device "discovery" or "enumeration" by executing a subset of the ARP. This is accomplished by issuing the "Prepare To ARP" command followed by repeatedly issuing the general "Get UDID" and "Assign Address" commands until no device acknowledges. Until the ARP process is complete the ARP Master must not wait more than two seconds before issuing a general "Get UDID" command after issuing the previous general "Get UDID" command. This restriction is important to allow another SMBus master to determine when it is safe to do an enumeration of the bus.

5.6.3.11. *ARP master behavior*



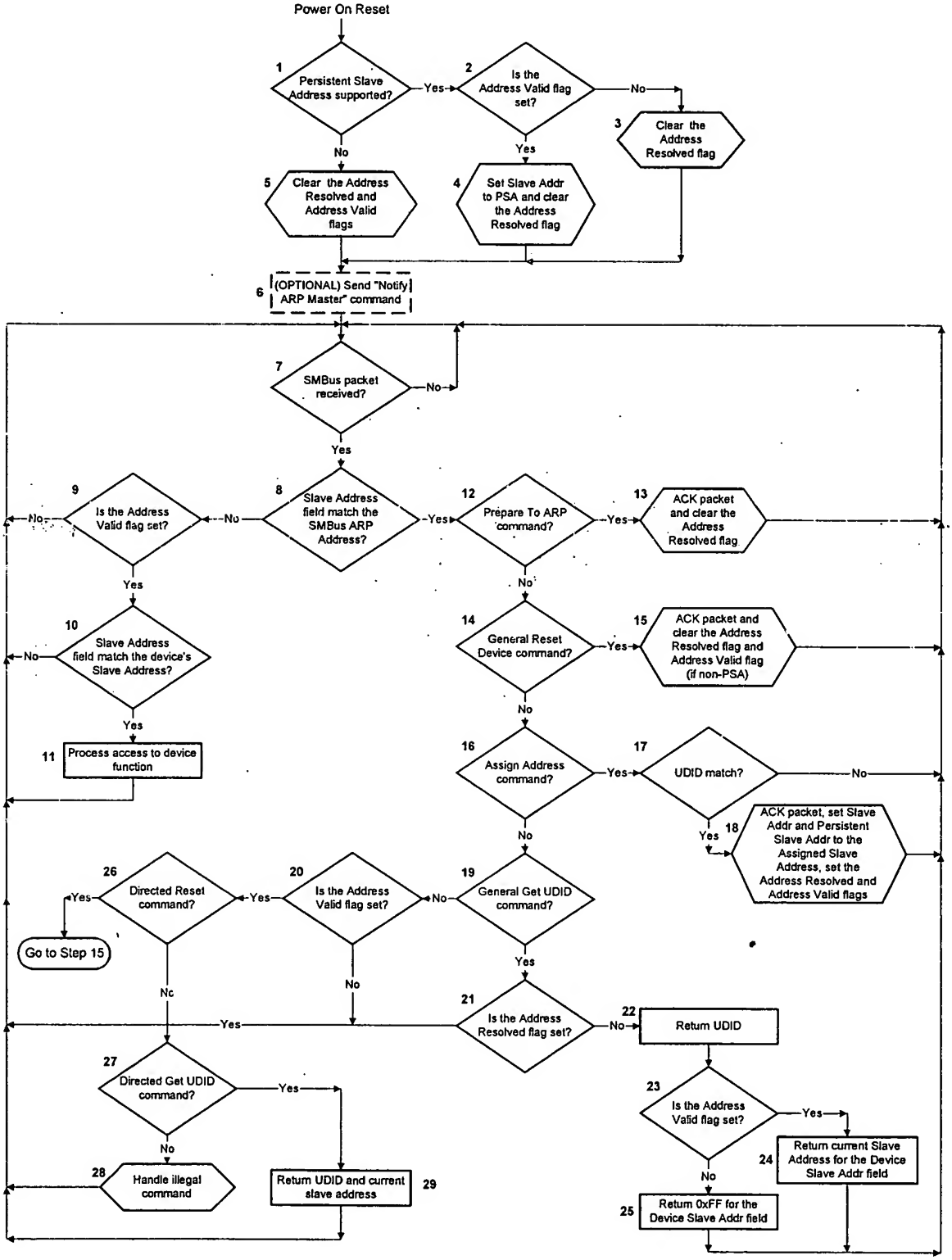
Referring to the previous flow diagram the ARP Master operates as follows:

1. Upon starting, the ARP Master will initialize its Used Address Pool. Initially this will consist of the slave addresses of fixed SMBus devices known to the ARP Master and reserved addresses (as defined in Appendix C).
2. Send the "Prepare To ARP" command.
3. Check for an acknowledgement for all bytes in the previous packet. If any bytes were not acknowledged then the ARP Master can assume that no ARP-capable devices are present and may therefore consider the ARP process complete and Proceed to step 4. If all bytes were acknowledged then go to step 6.
4. The ARP Master found no response to the "Prepare To ARP" command so it can assume that no ARP-capable devices are present in the system at this time. The ARP Master may periodically re-issue the "Prepare To ARP" command to discover any ARP-capable devices added. Proceed to step 5.
5. Wait for an SMBus packet. If a packet is received proceed to step 15.
6. Send the "Get UDID" command.
7. Check for an acknowledgement for the first three bytes and verify that the byte count value received is 0x11. If not, then the ARP Master can assume that an ARP-capable device(s) is no longer present and may therefore consider the ARP process complete and Proceed to step 4. Otherwise proceed to step 8.
8. Check the value of the Device Slave Address received. If 0xFF then proceed to step 11 since this device does not possess a valid slave address. Otherwise proceed to step 9.
9. Determine if this device has a fixed slave address. If bits 127 and 126 of the UDID are 00b than it has a fixed address, so procede to step 12. Otherwise proceed to step 10.
10. The device possesses a valid slave address. However, the ARP Master must check this address against the Used Address Pool to insure that no other device has already been assigned the same address. If the received Device Slave Address is found in the Used Address Pool then proceed to step 11. If not, then the device can keep its current slave address but needs acknowledgement from the ARP Master so proceed to step 12.
11. Select a slave address that is not in the Used Address Pool and proceed to step 12.
12. Send the "Assign Address" command with the UDID returned by the device in the "Get UDID" command packet.
13. Check for acknowledgement of all bytes in the "Assign Address" command packet. If any byte was not acknowledged then the ARP Master assumes the device is no longer present; proceed to step 6 to determine if there are more devices requiring address resolution. If all bytes were acknowledged then the ARP Master assumes that the device has accepted the address assignment; proceed to step 13.
14. The device now has a valid slave address. The ARP Master must add this address to the Used Address Pool. Proceed to step 6 to determine if there are more devices requiring address resolution.
15. The ARP Master checks to see if the received packet was the "Notify ARP Master" command. If so, then it must execute the ARP to resolve the address for the newly added device(s); proceed to step 6. If not, then proceed to step 16.
16. The ARP Master received a non-ARP related packet. Process it accordingly and proceed to step 5.

The ARP Master behavior flow diagram cover the case when the ARP Master has come out of a reset state. The ARP supports "hot-plug" devices so the ARP Master must be prepared to execute the ARP at any time; step 15 covers the case when a device a device issues the "Notify ARP Master" command. Since that command is optional the ARP Master cannot rely on a notification from the device upon its appearance on the SMBus. As such the ARP Master should periodically issue the "Prepare To ARP" command if the SMBus implementation supports "run-time" device addition. Doing so will also allow the ARP Master to determine if any ARP-capable devices have been removed from the SMBus. In this case the ARP Master can remove addresses from the Used Address Pool if it doesn't detect a response from a device previously assigned an address.

The flow diagram also does not cover consideration for bus timeout mechanisms or retries. These should be implemented in order to comply with the bus timing specifications.

5.6.3.12. ARP-capable device behavior



Referring to the previous flow diagram an ARP-capable device operates as follows:

1. After exiting the power on reset state, a device that supports the Persistent Slave Address (PSA) will go to step 2 to see if it is valid. If the device does not support the PSA, it will proceed to step 5.
2. A device supporting PSA must check its Address Valid flag which is a non-volatile. If that flag is set then it has previously received an assigned slave address and proceeds to step 4. If the Address Valid flag is cleared then it must proceed to step 3
3. Although the device supports the PSA the value is currently invalid. The device must clear the Address Resolved flag indicating that it has not had its slave address assigned. Proceed to step 6.
4. The device has a valid PSA so it assumes that slave address for now. However, this address has not been resolved by the ARP Master so the device must clear its Address Resolved flag. Proceed to step 6.
5. The device does not support the PSA so it must clear its Address Valid and Address Resolved flags. Proceed to step 6.
6. If supported, the device will master the SMBus and send the "Notify ARP Master" command. This will inform the ARP Master that a new device is present. Proceed to step 7.
7. The device waits for an SMBus packet.
8. Upon receipt of an SMBus packet the device must first check the received slave address against the SMBus Device Default Address. If there is a match then it proceeds to step 12, otherwise it proceeds to step 9.
9. The received address is not the SMBus Device Default Address so the packet is potentially addressed to the device's core function. The device must check its Address Valid bit to determine whether or not to respond. If the Address valid bit is set then it proceeds to step 10, otherwise it must return to step 7 and wait for another SMBus packet.
10. Since the device has a valid slave address it must compare the received slave address to its slave address. If there is a match then it proceeds to step 11, otherwise it must return to step 7 and wait for another SMBus packet.
11. The device has received a packet addressed to its core function so it acknowledges the packet and processes it accordingly. Proceed to step 7 and wait for another SMBus packet.
12. The device detected a packet addressed to the SMBus Device Default Address. It must check the command field to determine if this is the "Prepare To ARP" command. If so, then it proceeds to step 13, otherwise it proceeds to step 14.
13. Upon receipt of the "Prepare To ARP" command the device must acknowledge the packet and make sure its Address Resolved flag is clear in order to participate in the ARP process. Proceed to step 7 and wait for another SMBus packet.
14. The device checks the command field to see if the "Reset Device" command was issued. If so, then it proceeds to step 15, otherwise it proceeds to step 16.
15. Upon receipt of the "Reset Device" command the device must acknowledge the packet and make sure its Address Valid (if non-PSA) and Address Resolved flags are cleared. This will allow the ARP Master to re-assign all device addresses without cycling power. Proceed to step 7 and wait for another SMBus packet.
16. The device checks the command field to see if the "Assign Address" command was issued. If so, then it proceeds to step 17, otherwise it proceeds to step 19.
17. Upon receipt of the "Assign Address" command the device must compare its UDID to the one it is receiving. If any byte does not match then it must not acknowledge that byte or subsequent ones. If all bytes in the UDID compare then the device proceeds to step 18, otherwise it must return to step 7 and wait for another SMBus packet.

18. Since the UDID matched, the device must assume the received slave address and update its PSA, if supported. The device must set its Address Valid and Address Resolved flags that means it will no longer respond to the "Get UDID" command unless it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled. Proceed to step 7 and wait for another SMBus packet.
19. The device checks the command field to see if the "Get UDID" command was issued. If so, then it proceeds to step 21, otherwise it proceeds to step 20.
20. The device may be receiving a directed command. Directed commands must only be acknowledged by slaves with a valid address. If the address is not valid then ignore the packet and return to step 7 and wait for another SMBus packet. If the address is valid then proceed to step 26.
21. Upon receipt of the "Get UDID" command the device must check its Address Resolved flag to determine whether or not it should participate in the ARP process. If set then its address has already been resolved by the ARP Master so the device proceeds to step 7 to wait for another SMBus packet. If the AR flag is cleared then the device proceeds to step 22.
22. The device returns its UDID and monitors the SMBus data line for collisions. If a collision is detected at any time the device must stop transmitting and proceed to step 7 and wait for another SMBus packet. If no collisions were detected then proceed to step 23.
23. The device must now check its Address Valid flag to determine what value to return for the Device Slave Address field. If the AV flag is set then it proceeds to step 24, otherwise it proceeds to step 25.
24. The current slave address is valid so the device returns this for the Device Slave Address field (with bit 0 set) and monitors the SMBus data line for collisions (i.e. another device driving a "0" when this device is "driving" a "1." Proceed to step 7 and wait for another SMBus packet.
25. The current slave address is invalid so the device returns a value of 0xFF and monitors the SMBus data line for collisions. If the ARP Master receives the 0xFF value it will know that the device requires address assignment. Proceed to step 7 and wait for another SMBus packet.
26. Is this a directed "Reset Device" command? If so then proceed to step 15. Otherwise proceed to step 27.
27. Is this the "Directed Get UDID" command? If so, then proceed to step 29. Return the UDID information. If not, then proceed to step 28
28. The device has not received a valid command so it must handle the illegal command in accordance with SMBus rules for error handling. Proceed to 7 and wait for another SMBus packet.
29. Return the UDID information and current slave address, then proceed to 7 and wait for another SMBus packet.

The flow diagram does not cover consideration for bus timeout mechanisms. These should be implemented in order to comply with the bus timing specifications. If the device supports the "Notify ARP Master" command it may wish to consider implementing a timeout mechanism. This mechanism could cause the device to re-issue the "Notify ARP Master" command if the ARP Master doesn't respond within a particular time period.

The device decodes the two internal flags as described in the following table:

Address Valid (AV Flag)	Address Resolved (AR Flag)	Meaning
Cleared	Cleared	The device does not have a valid slave address and will participate in the ARP process. This is the POR state for a device that doesn't support the PSA or if it does it has not previously been assigned a slave address.
Cleared	Set	Illegal state!
Set	Cleared	The device has a valid slave address but must still participate in the ARP process.
Set	Set	The device has a valid slave address that has been resolved by the ARP Master. The device will not respond to the "General Get UDID" command. However, it could subsequently receive an "Assign Address" command and would change its slave address accordingly.

Table 8: Device decodes of AV and AR flags

5.6.3.13. Enumeration rules

Any device may enumerate the bus provided that the device is intelligent and capable of doing so. Additionally, the enumerating device must provide "snooping" capabilities to guarantee that multiple devices aren't enumerating/ARPing at the same time. An enumerator must adhere to the following rules:

1. If an enumerator sees a "Prepare to ARP" or "Get UDID" command it must immediately stop enumerating.
2. An enumerator must monitor the bus for ARP commands for at least 2 seconds before beginning the enumeration process with the "Prepare to ARP" command.
3. If an enumerator sees an unassigned address then it must issue a host notify command and stop enumerating.

5.6.3.14. Example scenarios

The ARP can be illustrated by the following examples. Note that the UDID values are simple examples for illustrative purposes. They do not necessarily represent legal values.

Example 1

In this scenario assume the following:

- The ARP Master has already exited its reset state and has run the ARP. The Used Address Pool does NOT contain the values 1001 000, 1001 001, ... 1001 111.
- New Device A has a UDID of 0x8123456789ABCDEF0000000000000000, does support the Persistent Slave Address and was previously assigned the slave address 1001 001. Its Address Valid flag is set but its Address Resolved flag is cleared.
- New Device B has a UDID of 0xF123456789ABCDE00000000000000000 and does not support the Persistent Slave Address. Its Address Valid and Address Resolved flags are cleared.
- New Device C has a UDID of 0xF123456789ABCDE10000000000000000 and does not support the Persistent Slave Address. Its Address Valid and Address Resolved flags are cleared.
- All devices exit their power on reset state at the same time.

The ARP will proceed as follows:

1. When the devices exit their power on reset state they will optionally attempt to issue the "Notify ARP Master" command. Assume for this example that all three devices do so.
2. All devices will transmit all bytes of the "Notify ARP Master" command without collision.
3. The ARP Master having received the "Notify ARP Master" command will issue the "Get UDID" command.
4. After detecting the repeat Start condition and receiving the SMBus Device Default Address with the R/W# bit set the devices will transmit the byte count for this command without collision. The devices will then begin to transmit their UDIDs as follows:

Device A: 1000...	1	0	
Device B: 1111...	1	1	
Device C: 1111...	1	1	
Seen on the bus:	1	0	Device A wins the bus arbitration

Device A wins the bus arbitration since it is transmitting a "0" as the 2nd msb in the most significant byte of the UDID whereas Devices B & C are transmitting "1." As such Devices B & C will cease transmission of this packet. Device A will complete its transmission.

1. The ARP Master will send the "Assign Address" command using Device A's slave address and UDID. Device A will then set its Address Resolved flag (the Address Valid flag was already set). Device A will no longer respond to the "Get UDID" command until it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled.
2. The ARP Master will add slave address 1001 001 to the Used Address Pool since Device A acknowledged the packet.
3. The ARP Master will issue the "Get UDID" command again.
4. Devices B & C having lost the previous arbitration will respond and will transmit the byte count for this command without collision.
5. The devices will begin to transmit their UDIDs. Since the UDIDs are equal through the first seven most significant bytes there will be no bus collisions. The eighth UDID byte will be transmitted as follows:

Device B: 1110 0000	1	1	1	0	0	0	0	0	
Device C: 1110 0001	1	1	1	0	0	0	0	1	
Seen on the bus:	1	1	1	0	0	0	0	0	Device B wins the bus arbitration

Device B wins the bus arbitration since it is transmitting a "0" as the last bit in the eighth byte of the UDID whereas Device C is transmitting "1." As such Device C will cease transmission of this packet. Device B will complete its transmission by sending the remaining eight bytes of the UDID and an 0xFF for the Device Slave Address field since its Address Valid bit is cleared.

1. Device B will await an assigned address since its Address Valid flag is cleared.
2. The ARP Master recognizes that the returned slave address field was 0xFF. It captures the returned UDID and selects an address (e.g. 1001 000) not in the Used Address Pool and issues the "Assign Address" command.

3. All devices will monitor the "Assign Address" command looking for a UDID match. Since Device B will match its UDID it will acknowledge the packet and adopt the slave address assigned by the ARP Master. Device B will also set its internal Address Resolved and Address Valid flags and will no longer respond to the "Get UDID" command until it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled.
4. The ARP Master will add slave address 1001 000 to the Used Address Pool since Device B acknowledged the packet.
5. The ARP Master will issue the "Get UDID" command again.
6. Device C having lost the previous arbitration will respond and will transmit the byte count for this command without collision. Since it is now the only device responding all remaining bytes will be transmitted without collision.
7. Device C will await an assigned address since its Address Valid flag is cleared.
8. The ARP Master recognizes that the returned slave address field was 0xFF. It captures the returned UDID and selects an address (e.g. 1001 010) not in the Used Address Pool and issues the "Assign Address" command.
9. All devices will monitor the "Assign Address" command looking for a UDID match. Since Device C will match its UDID it will acknowledge the packet and adopt the slave address assigned by the ARP Master. Device C will also set its internal Address Resolved and Address Valid flags and will no longer respond to the "Get UDID" command until it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled.
10. The ARP Master will add slave address 1001 010 to the Used Address Pool since Device C acknowledged the packet.
11. The ARP Master will issue the "Get UDID" command again.
12. Since all three devices have their internal Address Resolved flag set they will not respond.
13. The ARP Master will detect that no device has acknowledged the packet and will terminate the ARP.

Example 2

In this scenario assume the following:

- The system is in the S5 state.
- The system does not contain any devices with slave address values 1001 000, 1001 001, ... 1001 111.
- Device A has a UDID of 0x0123456789ABCDEF0000000000000000, does support the Persistent Slave Address and was previously assigned the slave address 1001 001. Device A was present in the system before it entered the S5 state. Its Address Valid flag is set but its Address Resolved flag is cleared.
- New Device B has a UDID of 0xFEDCBA98765432100000000000000000, does support the Persistent Slave Address and was previously assigned the slave address 1001 001 when present in another system. Device B was added to the system while it was in the S5 state. Its Address Valid flag is set but its Address Resolved flag is cleared.
- Both devices exit their power on reset state at the same time.

The ARP will proceed as follows:

1. The system transitions to the S0 state (assume the user pressed the power button).
2. The ARP Master will exit the reset state and will initialize its Used Address Pool.
3. When the devices exit their power on reset state they may attempt to issue the "Notify ARP Master" command; assume that they do for this example. The ARP Master will attempt to issue the "Prepare To ARP" command.

4. If the ARP Master receives the "Notify ARP Master" command before it can issue the "Prepare To ARP" command it will simply ignore it.
5. The ARP Master will issue the "Get UDID" command since presumably the "Prepare To ARP" command was acknowledged.
6. After detecting the repeat Start condition and receiving the SMBus Device Default Address with the R/W# bit set the devices will transmit the byte count for this command without collision.
7. The devices will begin to transmit the most significant byte of their UDID as follows:

Device A: 0000 0001	0	
Device B: 1111 1110	1	
Seen on the bus:	0	Device A wins the bus arbitration

Device A wins the bus arbitration since it is transmitting a "0" as the msb in the most significant byte of the UDID whereas Device B is transmitting "1." As such Device B will cease transmission of this packet. Device A will complete its transmission.

1. The ARP Master will send the "Assign Address" command using Device A's slave address and UDID. Device A will then set its Address Resolved flag (the Address Valid flag was already set). Device A will no longer respond to the "Get UDID" command until it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled.
2. The ARP Master will add slave address 1001 001 to the Used Address Pool since Device A acknowledged the packet.
3. The ARP Master will issue the "Get UDID" command again.
4. Device B having lost the previous arbitration will respond and will transmit the byte count for this command without collision. Since it is now the only device responding all remaining bytes will be transmitted without collision
5. The ARP Master recognizes that the returned slave address was already in the Used Address Pool. It captures the returned UDID and selects an address (e.g. 1001 000) not in the Used Address Pool and issues the "Assign Address" command.
6. All devices will monitor the "Assign Address" command looking for a UDID match. Since Device B will match its UDID it will acknowledge the packet and adopt the new slave address assigned by the ARP Master. Device B will keep its internal Address Valid flag set and will set its Address Resolved flag so that it will no longer respond to the "Get UDID" command until it receives the "Prepare To ARP" or "Reset Device" commands or is power cycled.
7. The ARP Master will add slave address 1001 000 to the Used Address Pool since Device B acknowledged the packet.
8. The ARP Master will issue the "Get UDID" command again.
9. Since both devices have their internal Address Resolved flag set they will not respond.
10. The ARP Master will detect that no device has acknowledged the packet and will terminate the ARP.

Appendix A – Optional SMBus signals

SMBSUS#

An **optional** third signal, SMBSUS#, goes low when the system enters the Suspend Mode. Suspend Mode refers to a low-power mode where most devices are stalled or powered down. Upon resume, the SMBSUS# returns high. The system then returns all devices to there operational state.

The SMBSUS# signal is included for clarity and completeness since many of the functions served by the System Management Bus relate to suspend and resume of the system.

The system can use the SMBCLK and SMBDAT lines to program device behavior. During normal operating mode the system may issue configuration commands to different devices. Those commands may tell the device how it is supposed to behave whenever the SMBSUS# line goes active. For example, the system might tell a power plane switcher to turn off power to the hard drive but keep the keyboard controller on when the system goes into suspend mode.

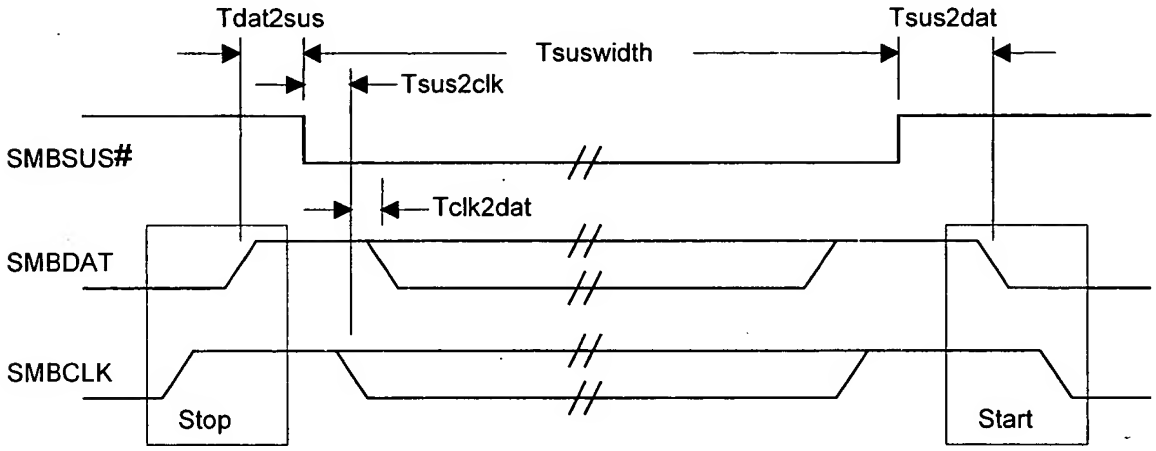


Figure A- 1: SMBus During Suspend

Timing	Min	Typical
$T_{DAT2SUS}$	0ns	tens of ms
$T_{sus2clk}$	0ns	tens of ns
$T_{clk2dat}$	0ns	0ns
$T_{suswidth}$		minutes, hours, weeks
$T_{sus2dat}$	0ns	hundreds of ms

Table 9: SMBus suspend parameters

SMBSUS# is not a wired-OR signal. It is an output from the device that controls system management functions, and an input to everything else.

During suspend there is no activity on the System Management Bus unless the SMBus is used to resume from suspend mode. Activity resumes after coming out of suspend.

Anytime after a STOP condition the SMBSUS# signal may go active low signifying the system is going into Suspend Mode. This can happen immediately (min = 0ns), but will probably take much longer. In fact, the final SMBus message might terminate minutes or hours before SMBSUS# goes low. Suspend

Mode could last a couple of seconds, minutes, hours, or weeks. Before the System Management Bus can send another message the system must come out of Suspend Mode, a process known as Resume. The resume process probably has to supply voltage to the System Management Bus anyway, although the SMBus may be awake during suspend. The resume process can take a long time, perhaps hundreds of milliseconds. Careful power-down sequencing of the SMBCLK and SMBDAT pull-ups will prevent devices from falsely seeing a START condition on the bus.

If power is supplied to the System Management Bus during suspend, a device may use it to awaken the system. The host or another device will watch for a START condition on the bus. That device initiates the resume sequence. Communication on the bus resumes when the system is out of the suspend state.

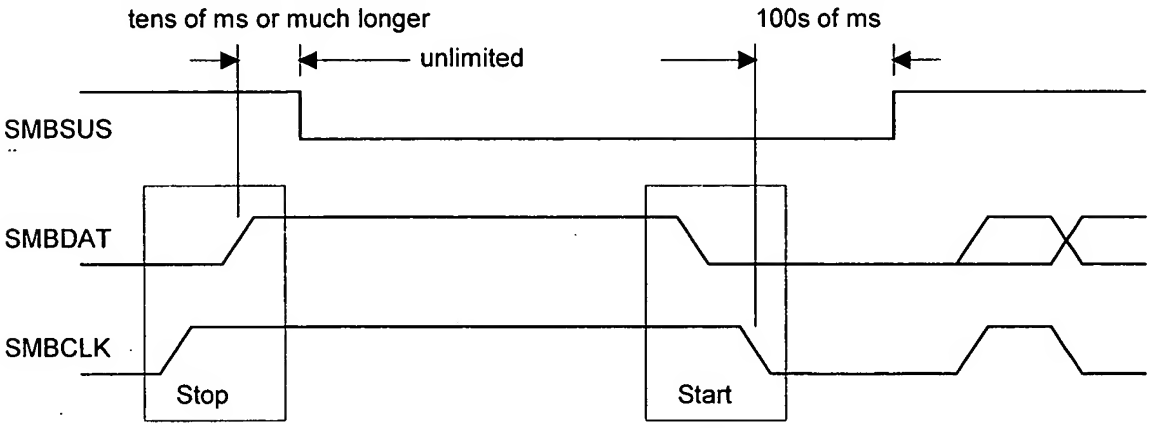


Figure A- 2: Using SMBus to Resume from Suspend

Since the SMBSUS# is an optional signal, some system devices may not know if the system is in suspend mode or not. Such a device may assume that if both SMBCLK and SMBDAT lines are high that the bus is alive and active. The possibility exists that this device may try to send a critical message to another device that accepts the SMBSUS# signal and is therefore asleep. Therefore it is important that a system is able to resume on a START condition if a non-suspendable master resides on the System Management Bus and that master can send critical messages to suspended devices.

SMBALERT#

Another **optional** signal is an interrupt line for devices that want to trade their ability to master for a pin. SMBALERT# is a wired-AND signal just as the SMBCLK and SMBDAT signals are. SMBALERT# is used in conjunction with the SMBus General Call Address. Messages invoked with the SMBus are 2 bytes long.

A slave-only device can signal the host through SMBALERT# that it wants to talk. The host processes the interrupt and simultaneously accesses all SMBALERT# devices through the Alert Response Address (ARA). Only the device(s) which pulled SMBALERT# low will acknowledge the Alert Response Address. The host performs a modified Receive Byte operation. The 7 bit device address provided by the slave transmit device is placed in the 7 most significant bits of the byte. The eighth bit can be a zero or one.

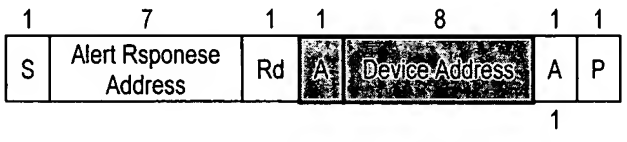


Figure A- 3: A 7-bit-Addressable Device Responds to an ARA

Figure A- 4: A 7-bit-Addressable Device Responds to an ARA with PEC

If more than one device pulls SMBALERT# low, the highest priority (lowest address) device will win communication rights via standard arbitration during the slave address transfer.

After acknowledging the slave address the device must disengage its SMBALERT# pulldown. If the host still sees SMBALERT# low when the message transfer is complete, it knows to read the ARA again.

A host which does not implement the SMBALERT# signal may periodically access the ARA.

Appendix B – Differences between SMBus and I²C

While SMBus is derived from I²C, there are several major differences between the specifications of the two busses in the areas of electricals, timing, protocols and operating modes.

DC specifications for SMBus and I2C

Both I²C and SMBus are capable of operating with mixed devices that have either fixed input levels (such as Smart Batteries) or input levels related to V_{DD}. When mixing devices, the I²C specification defines the V_{DD} to be 5.0 Volt +/- 10% and the fixed input levels to be 1.5 and 3.0 Volts.

Instead of relating the bus input levels to V_{DD}, SMBus defines them to be fixed at 0.8 and 2.1 Volts. This SMBus specification allows for bus implementations with V_{DD} ranging from 3 to 5 Volts +/- 10%. SMBus has relaxed the initial requirement for fixed input levels of 0.6 and 1.4 Volts, in order to reduce the cost of SMBus compliant devices. Devices compliant with the 1.0 specification of SMBus will still operate with higher versions of SMBus.

A second difference in the DC parameters between I²C and SMBus is in the power consumption of the low-power version of the bus. SMBus low-power electricals are designed to accommodate extremely low power consumption devices, such as the control circuitry within a Smart Battery. These devices have limited current sinking capabilities and a low power consumption bus is essential for maintaining communications without draining the battery of a mobile computer. As a result, SMBus sets more stringent DC requirements than I²C. One of the main differences is the IOL specification for VOL = 0.4 Volts. SMBus low-power devices are required to sink a minimum of 100 uA as opposed to 3mA specified for I²C devices for the same VOL.

A third difference is in the specification of the maximum leakage current for each device connected to the bus. I²C specifies the maximum leakage current to be 10 uA. SMBus version 1.0 specified maximum leakage current of 1 uA. Version 1.1 of the SMBus specification relaxes the leakage requirements to 5 uA, in order to reduce the cost of testing of SMBus devices.

Finally, SMBus does not specify a maximum bus capacitance. Instead it specifies the $I_{PULLDOWN}$ maximum of 350 uA. Bus capacitance can be calculated taking into consideration the maximum rise time and $I_{PULLDOWN}$.

The following table lists the main differences among the DC parameters for I²C and SMBus.

DC parameter comparison between Standard I²C, Fast I²C and SMBus devices								
Symbol	Parameter	Std I²C mode device		Fast I²C mode device		SMBus device		Units
		MIN	MAX	MIN	MAX	MIN	MAX	
V_{IL}	Fixed input level	-0.5	1.5	-0.5	1.5	-	0.8	V
	V_{DD} related input level	-0.5	$0.3V_{DD}$	-0.5	$0.3\ V_{DD}$	N/A	N/A	V
V_{IH}	Fixed input level	3.0	V_{DD}max+ 0.5	3.0	V_{DD}max+0 .5	2.1	5.5	V
	V_{DD} related input level	$0.7V_{DD}$	V_{DD}max+ 0.5	$0.7V_{DD}$	V_{DD}max+0 .5	N/A	N/A	V
V_{HYS}	V_{IH}-V_{IL}	N/A	N/A	$0.05V_{DD}$	-	N/A	N/A	V
V_{OL}	V_{OL} @ 3mA	0	0.4	0	0.4	N/A	N/A	V
	V_{OL} @ 6mA	N/A	N/A	0	0.6	N/A	N/A	
	V_{OL} @ 350uA	N/A	N/A	N/A	N/A	-	0.4	
I_{PULLUP}		N/A	N/A	N/A	N/A	100	350	uA
I_{LEAK}		-10	10	-10	10	-5	5	uA

Table 10: DC parameter differences between SMBus and I2C

Timing specification differences between SMBus and I^2C

There are differences in the timing specifications between I^2C and SMBus. As in the case of DC specification, proper understanding of the parameters is needed in order to combine reliably I^2C with SMBus devices.

SMBus defines a minimum bus clock frequency FSMB of 10 KHz. I^2C does not specify any minimum bus frequency. Besides maintaining effective bus throughput, this SMBus specification parameter can be used as a simple way to detect a bus idle condition (in addition or in lieu of detecting each STOP condition) as well as to implement bit timeout.

SMBus defines a data hold time, the time during which SMBDAT must remain valid from the falling edge of SMBCLK, of 300 nS. I^2C defines this hold time as zero.

Maximum clock frequency for SMBus is defined at 100 KHz. I^2C provides two modes of operation. The STANDARD MODE up to 100 KHz and the FAST-MODE up to 400 KHz.

SMBus defines a clock low time-out, TTIMEOUT of 35 ms. I^2C does not specify any timeout limit.

SMBus specifies TLOW: SEXT as the cumulative clock low extend time for a slave device. I^2C does not have a similar specification.

SMBus specifies TLOW: MEXT as the cumulative clock low extend time for a master device. Again I^2C does not have a similar specification.

SMBus defines both rise and fall time of bus signals. I^2C does not.

The SMBus time-out specifications do not preclude I^2C devices co-operating reliably on the SMBus. It is the responsibility of the designer to ensure that I^2C devices are not going to violate these bus timing parameters.

Other differences

ACK and NACK usage

There are the following differences in the use of the NACK bus signaling:

In I^2C, a slave receiver is allowed not to acknowledge the slave address, if for example is unable to receive because it's performing some real time task. SMBus requires devices to acknowledge their own address always, as a mechanism to detect a removable device's presence on the bus (battery, docking station, etc.)

I^2C specifies that a slave device, although it may acknowledge its own address, some time later in the transfer it may decide that it cannot receive any more data bytes. The I^2C specifies, that the device may indicate this by generating the not acknowledge on the first byte to follow.
Besides to indicate a slave device busy condition, SMBus is using the NACK mechanism also to indicate the reception of an invalid command or data. Since such a condition may occur on the last byte of the transfer, it is required that SMBus devices have the ability to generate the not acknowledge after the transfer of each byte and before the completion of the transaction. This is important because SMBus does not provide any other resend signaling. This difference in the use of the NACK signaling has implications on the specific implementation of the SMBus port, especially in devices that handle critical system data such as the SMBus host and the SBS components.

SMBus protocols

Each message transaction on SMBus follows the format of one of the defined SMBus protocols. The SMBus protocols are a subset of the data transfer formats defined in the I^2C specifications. I^2C devices that can be accessed through one of the SMBus protocols are compatible with the SMBus specifications. I^2C devices that do not adhere to these protocols cannot be accessed by standard methods as defined in the SMBus and ACPI specifications.

Appendix C – SMBus device address assignments

The following table represents the SMBus device assignments as of March 31, 1999.

Slave Address	Description	Specification
0001 000	SMBus Host	System Management Bus Specification[1] v 1.1 December 1998
0001 001	Smart Battery Charger	Smart Battery Charger Specification[1] v 1.1 December 1998
0001 010	Smart Battery Selector Smart Battery System Manager	Smart Battery Selector Specification[1] v 1.1 December 1998 Smart Battery System Manager Specification[1] v 1.0B August 1999
0001 011	Smart Battery	Smart Battery Data Specification[1] v 1.1 December 1998
0001 100	SMBus Alert Response	System Management Bus Specification[1] v 1.1 December 1998
0101 000	ACCESS.bus host	
0101 100	Reserved by previous versions of the SMBus specification for LCD Contrast Controller. This address may be reassigned in future versions of the SMBus specification.	
0101 101	Reserved by previous versions of the SMBus specification for CCFL Backlight Driver. This address may be reassigned in future versions of the SMBus specification.	
0110 111	ACCESS.bus default address	
1000 0XX	Reserved by previous versions of the SMBus specification for PCMCIA Socket Controllers (4 addresses) . These addresses may be reassigned in future versions of the SMBus specification.	
1000 100	Reserved by previous versions of the SMBus specification for (VGA) Graphics Controller. This address may be reassigned in future versions of the SMBus specification.	
1001 0XX	Unrestricted addresses	System Management Bus Specification[1] v 1.1 December 1998
1100 001	SMBus Device Default Address	System Management Bus Specification[1] v 1.1 December 1998

Table 11: Assigned SMBus addresses

[1] - Available from the SBS-IF at www.sbs-forum.org

The I²C-Faq

To Main Page

Questions & Answers

Table of Contents

The I2C FAQ 2.0 History of the I2C Bus

The I2C bus was developed in the early 1980's by Philips semiconductors.It's purpose was to provide an easy way to connect a CPU to peripherial chips in a TV-set.

Normal Computer systems use ByteWide buses to accomplish this task. This results in lots of copper tracks on PCB's to route the Address and datalines. Not to mention a bunch of address decoders and glue logic to connect everything . In mass production items such as TV-sets, VCR's and audio equipment this is not acceptable . In these appliances every component counts. And a component sless means more money for the producer and cheaper products for the customer .

Furthermore lots of control lines implies that the systems is more susceptible to disturbances by EMC and ESD . The research done by Philips Labs in Eindhoven (The Netherlands) resulted in a 2 wire communication bus called the I2C bus.

I2C is an acronym for Inter-IC bus . It's name literally explains it's purpose: to provide a communication link between Integrated Circuits.

Nowadays the extent of the bus goes much further than Audio and Video equipment.The bus is generally accepted in industry . Offsprings like D2B and ACCESS bus find their ways into computer peripherals like keyboards,mice,printers,monitors,etc... . The I2C BUs has been adopted by several leading chip manufacturers like Xicor,SGS-Thomson,Siemens, Intel,TI,Maxim,Atmel,Analog Devices.

The I2C FAQ 2.0

The I2C Bus Protocol

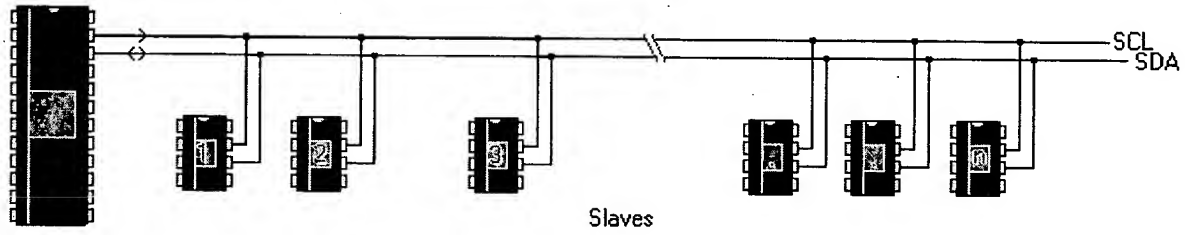
The BUS physically consists of 2 active wires and a ground connection. The active wires ,SDA en SCL,are both bidirectional . Where SDA is the Serial DAta line and SCL is the Serial CLock line.

Every component hooked up to the bus has its own unique address whether it is a CPU,LCD driver,memory,or complex function chip . Each of these chips can act as a receiver and/or transmitter depending on it's functionality . Obviously an LCD driver is only a receiver , while a memory or I/O chip can both be transmitter and receiver. Furthermore there may be one or more BUS MASTER's.

The BUS MASTER is the chip issuing the commands on the BUS . In the I2C protocol specification it is stated that the IC that initiates a data transfer on the bus is considered the BUS MASTER. At that time all the others are regarded to as the BUS SLAVE .

As mentioned before , the IC bus is a Multi-MASTER BUS. This means that more than one IC capable of initiating data transfer can be connected to it. As MASTERs are generally microcomputers let's take a look at a general 'inter-IC chat' on the bus.

Lets consider the following setup :



Case : The CPU wants to talk to one of it's slaves.

The CPU will issue a START condition (see further on for description of all these conditions) This acts as an 'ATTENTION' signal to all of the connected ic's.All IC's on the bus will listen to the bus for incoming data.

Then the CPU sends the address of the device he wants to access . This takes 8 clock pulses. At this moment in time all IC's will compare this address with their own . If it doesn't match they simply do nothing and wait until the bus is released by the STOP condition . If the address matches however the chip will produce a responce called the ACKNOWLEDGE signal .

If the cpu gets this ACKNOWLEDGE then it can start transmitting or receiving data . In our case the CPU will transmit data.When all is done the CPU will issue a STOP condition. This is a signal that the bus has been released and that the IC's may expect another transmission to start any moment.

We have had several states on the BUS right now : START, address ,ACKNOWLEDGE ,DATA ,STOP. These are all unique conditions on the BUS . Before we take a closer look into these bus conditions we need to understand a bit about the physical structure and

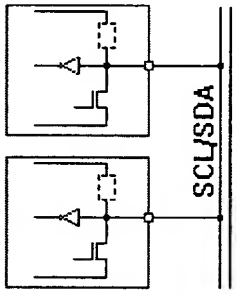
hardware of the bus

The I2C FAQ 2.0

The I2C Bus hardware structure

As explained before the bus physically consists of 2 active wires and a ground connection. The active wires ,**SDA** en **SCL**,are both bidirectional . Where SDA is the Serial **DA**ta line and SCL is the Serial **CL**ock line.

Both of these lines are initially bidirectional . This means that these lines can be driven by the chip or from the outer world . To avoid 'the fried chip' effect these signals use open collector or open drain (depending on the technology) outputs.



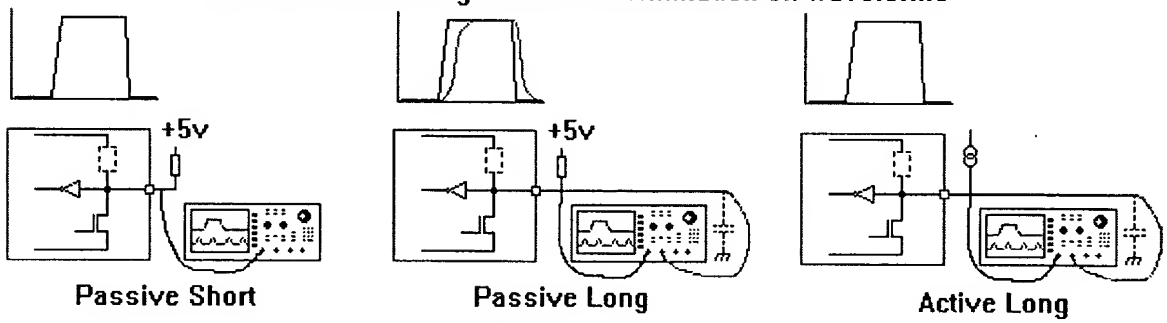
The bus interface is built around an input buffer and an open drain or open collector transistor . When nothing happens the bus lines are in the logic HIGH state . Note here that an external PULL-UP resistor is neccessary . It is a very common error to forget these pull-up resistors . To put something on the BUS the chip drives its output transistor , thus pulling the BUS to a LOW level . When the bus is IDLE (nothing going on) both lines are HIGH

The pull up resistor in the devices is actually a small current source or even inexistent..

The nice thing about this concept is that it has a 'built-in' bus mastering technique. If the bus is 'occupied' by a chip that is sending a 0 then all other chips loose their comm's capability. More will be explained about this further on in the Faq.

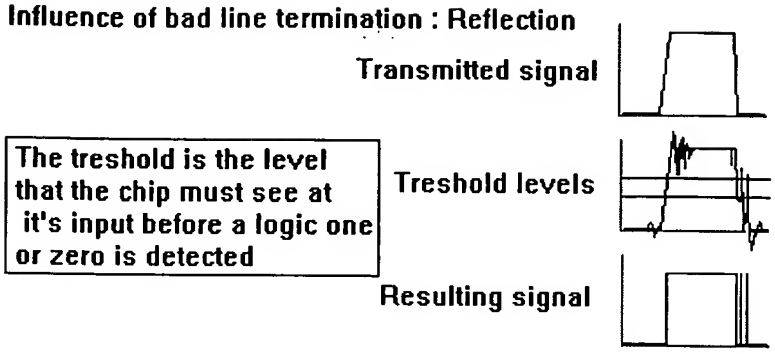
However the open collector technique has a drawback too. If you have a long bus this will have a serious effect on the speed you can obtain . Long lines present a capacitive load onto the output . Since the pull-up is passive you are facing an RC constant which will reflect onto the shapes of the signals. The higher this RC constant the slower you can go. this is due to the effect that it influences the 'sharpness' of the edges on the I2C bus . At a certain point the chip will not be able to distinguisch clearly between a logic 1 and 0.

Influence of line length and bus termination on waveforms



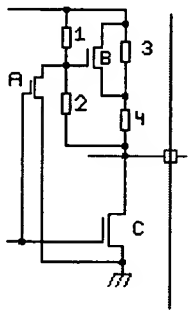
What's more is that you can get reflections at high speed .This can be so bad that 'ghost signals' disturb your transmission and corrupt the data you transmit. And then even the Shmitt trigger at the input of the chip will not keep you out of trouble

Influence of bad line termination : Reflection



Therefore some strict electrical specs have been set forth. More about these in the next chapter.

To overcome this problem , Philips has developed an Active I2C terminator. This consists of twin charge pumps. You can look at this device as a dynamic resistor. The moment the state changes it provides a large current (low dynamic resistance) to the bus. Doing this it can charge the parasitic capacitor very quickly. Once the voltage has risen above a certain level the high current mode cuts out and the output current drops sharply.

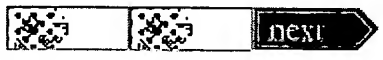


As long as the bus is kept low (transistor C is ON) the charge pump is disabled because the gate of transistor B is kept low by transistor A.

The moment the chip releases the bus (A and C turn off) the capacitor will start charging .Thus drawing current trough all four of the resistors (1234) The voltage drop over resistor 2 will cause the transistor B to turn on Thus shorting out resistor 3. Since Resistor 3 is a relatively low value the current will rise . At a certain point in time the drop between Gate and Source will not be big enough to keep the transistor B turned on. It will then turn-off and the charge injection will stop.At that time only the pull up resistor remains to overcome the charge leakage in the bus.

OPf course the above is a simple explanation. The actual device implements more circuitry. To prevent 'overcharging' when another chip is still keeping the bus low.

This device can come in handy if you need to overcome some meters of cabling.

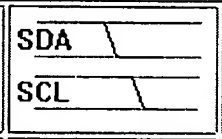
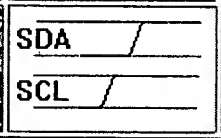


The I2C FAQ 2.0

I2C Bus Events: The Start and Stop conditions

Prior to any transaction on the bus a START condition needs to be issued on the bus.The start condition acts as a signal to all connected IC's that something is about to be transmitted on the BUS . As a result , all connected chips will listen to the bus.

After a message has been completed a STOP condition is sent . This is the signal for all devices on the bus that the bus is 'free again'. The chip that was accessed during the message will now process the information received (if not processed during the reception of the message).

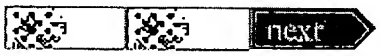
Start		The chip issuing the Start condition first pulls the SDA (data) line low. and next pulls the SCL (clock) line low.
Stop		The Bus MASTER first releases the SCL and then the SDA line.

A few notes about this start and stop conditions.

Further on in this Faq we will talk about data telegrams . You will see there that a single message can contain multiple Start conditions.

The use of this so called 'repeated start' is common in I2C .

A Stop condition ALWAYS denotes the END of a transmission. Even if it is issued in the middle of a transaction or in the middle of a byte . It is 'good behaviour' for a chip that ,in this case , it disregards the information sent and resumes to the 'listening state'.

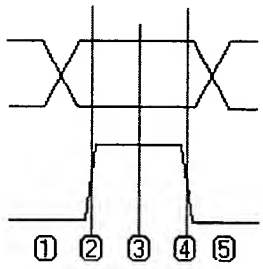


The I2C FAQ 2.0

I2C Bus Events: receiving a byte from a slave

Once the slave has been adressed and , the slave has acknowledged this ,a byte could be received from the slave. That is if the R/W bit in the adress was set to READ.

The protocol syntax is the same as in sending a byte to the slave. Except that now , the master is not allowed to touch the SDA line. Prior to sending the first of 8 clock pulses on the SCL line , the MASTER releases the SDA line .The slave will now take control of this line.The line will then go high or , if the slave wants to send a 0 , remain low .

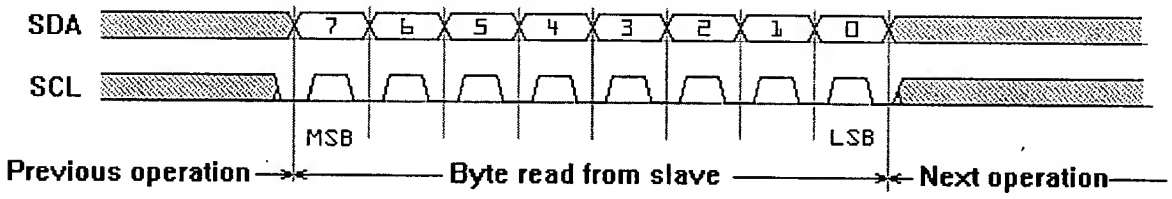


All the master has to do is raise the SCL line (2), read the level on SDA (3) and lower the SCL line (4). The slave will not change the data during the time that SCL is high.

(Obvious. Otherways it could cause a Start or Stop condition !)

During the times 1 and 5 the slave may change the state of the SDA line.

In total this sequence has to be done 8 times to complete the byte . Bytes are **always** transmitted **MSB first** .



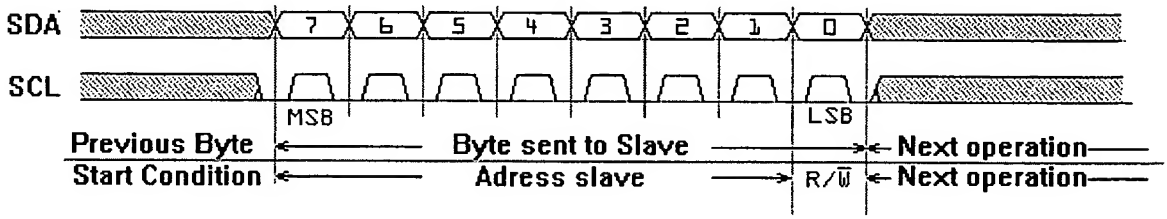
The meaning of all bytes beeing read is slave dependant.There is no such thing as a 'universal status register' . You need to consult the data sheet of the slave beeing adressed to know the meaning of each bit in any byte transmitted.

The I2C FAQ 2.0

I2C Bus Events: sending a byte to a slave

Once the start condition has been given a byte needs to be transmitted by the MASTER to the SLAVE.

This first byte after a START will identify the slave on the bus and select the mode of operation.The meaning of all following bytes is slave dependant .



A number of adresses have been reserved for special purposes. These will be detailed on later in the Faq . On of these adresses is reserved for the 'extended Adressing Mode' . As the bus gained in popularity they soon discovered that the number of free adresses was too small. Therefore on of these 'reserved adresses' has been allocated to a new task. If a normal (not 'extended adressing' capable) slave receives this adress it will do nothing (since it's not his adress) .

If there are slaves on the bus that have this mode they will ALL respond to the ACK cycle issued by the master. (more about the ACK later on). The second byte that gets transmitted by the master will then be taken in and evaluated against their adress .

Statement:

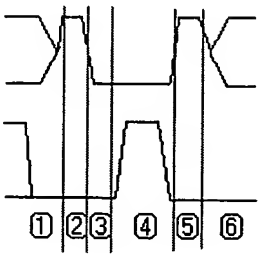
Bit 0 of the first byte after the Start condition determines the slave access mode

This statement still stands in this 10 bit 'extended adressing ' mode !.

The I2C FAQ 2.0 **I2C Bus Events: Giving acknowledge**

Upon reception of a byte from the slave , the MASTER must acknowledge this to the slave device .

The master is in full control of the SDA and the SCl line.



After transmission of the last bit to the master (1) the slave will release the SDA line.

The SDA line should then go high (2). The Master will now pull the SDA line low (3) .

Next the master will give a clock pulse on the SCL line (4). After completion of this clockpulse the master will again release the SDA line.(5).

The slave wil now regaing control of the SDa line.(6)

Note:

The above waveform is slightly exagerrated. You wiill not notice the going high of the waveform.A small spike might barely be visible.

Statement

*An **acknowledge** of a byte received from a slave is **always** necessary **EXCEPT on the last byte received.***

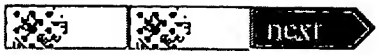
If the master wants to stop receiving data from the slave it must be able to send a stop condition.

Since the slave reagins control of the SDA line after the ACk.cycle issued by the master this could lead to problems.

Suppose the next bit that is ready to be sent to the master is a 0. The SDA line would be pulled low by the slave immediately after the master takes the SCL line low. The master now attempts to make a Stop appear on the bus. It raises the SCL lione first and then tries to raise the SDA line ... but this one is held low by the slave. Conclusion :No Stop has been present on the bus.

This condition is called a NACK : Not ACKnowledge . Do not confuse this with No ACKnowledge !

Condition	**Can only occurr...**
Not acknowledge	after a **MASTER** has **read** a byte **from** a **slave**
No acknowledge	after a **MASTER** has **written** a byte **to** a **slave**

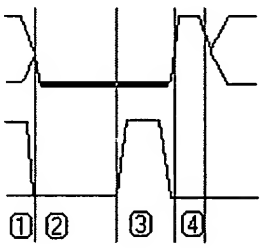

next

The I2C FAQ 2.0

I2C Bus Events: Getting acknowledge

When an adress or data byte has been transmitted onto the bus then this must be ACKNOWLEDGED by the slave(s) .In case of an adress : If the address matches it's own then that slave and only that slave will respond to the with an ACK . In case of a byte transmitted to an already adressed slave then that slave will respond with an ACK.

The slave that is going to give an ACK pulls the SDA line immediately after reception of the 8th bit transmitted or , in case of an adress byte ,immediately after evaluation of it's adress. In practical applications this will not be noticable.



This means that , the moment the MASTER takes SCL low to complete the transmission of the bit (1), SCL will be pulled low by the SLAVE.(2)

The master now Issues a clock pulse on the SCL line(3). the slave will release the SDa line upon completion of this clockpulse.(4)

The bus is now free again for the master to continue sending data , or generating a stop condition.

In case of data beeing written to a slave this cycle must be completed before a stop can be given.The Slave will be blocking (SDA kept low by slave) the bus until the master has given a clockpulse on the SCL line.

next

The I2C FAQ 2.0 **I2C Bus Events: No Ack**

This is not exactly a condition. It is merely a state in the data flow between master and slave.

If , after transmission of the 8 th bit from the MASTER to the slave , the slave does not pull the SDA line low, then we have a No ACK condition .

This means that either :

- The slave is not there (in case of an address)
- The slave missed a pulse and got out of sync with the SCL line of the master.
- The bus is 'stuck' . One of the lines could be held permanenently low.

In any case the MASTER should abort by attempting to send a STOP condition on the bus.

A test for 'Stuck bus' can be performed in the STOP cycle.

The I2C FAQ 2.0

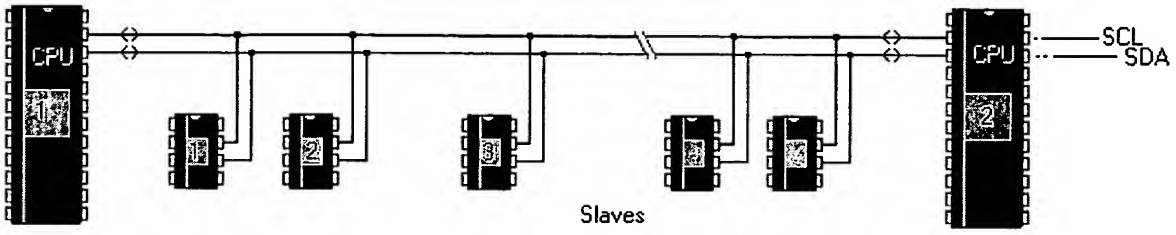
Arbitration on the bus

So far we have seen the operation of the bus from the masters point of view and using only one master on the bus.

The I2C bus was originally developed as a multimaster bus. This means that more then one initiator can be active in the system.

When using only one master on the bus there is no real risk of corrupted data , except if a slave device is malfunctioning or if there is electrically something wrong with the bus.

That situation changes with 2 CPU's.



When CPU 1 issues a START and sends an address all slaves (including CPU2 which at that time is considered a slave too) will listen . If the address does not match the address of CPU2 he has to wait until the bus is free. (STOP condition). So far no problem .

As long as the two cpu's monitor what is going on on the bus (start and stop) and that they know a transaction is going on because the last issued command was not a STOP there is no problem.

But as Murphy is, as usual , short at hand... It's when you least expect it that it goes wrong.

Suppose one of the CPU's missed the START condition and still thinks nothing is going on ... , or it came just out of reset and wants to start talking on the bus. These are real-life cases. Consider a multimaster system that has just been powered up. You will not know which cpu will come out of reset first. (in most cases you don't) .

This could lead to big problems .

How can you know if some other device is sending on the bus ?

Fortunately the physical BUS setup helps us out here. Since the bus structure is a wired AND (if one device pulls a line low it stays low) you can test for bus occupation.

When you (as a MASTER) change the state of a line to HIGH , you MUST always check that it has gone to HIGH. If it stays low then : BACKOFF ! it's occupied ! . some other device is pulling the line low.

so the general rule of thumb is : If you can't get a certain line high : Backoff and wait until a stop condition is seen before resuming . Face the hard reality : you just lost arbitration of the bus.

This backoff condition has to be maintained until a valid STOP condition has been seen on the bus . Then and only then an attempt can be made to start talking again .

One problem down .. one to go :

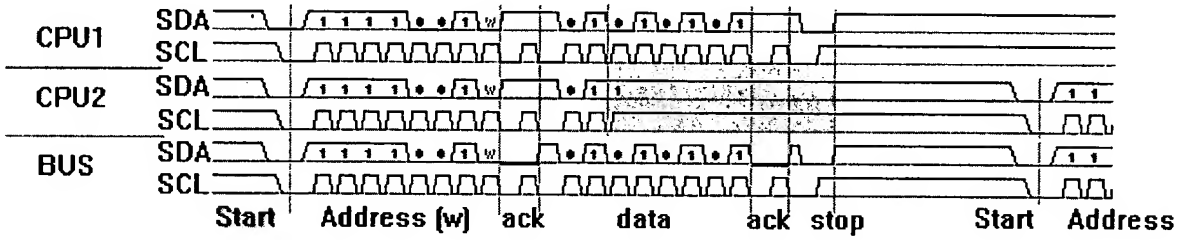
What about the risk of data corruption ?.

Well .. here again the physical bus structure helps us out.

Since the 1st rule says that you loose arbitration when you cannot raise either SCL or SDA when needed this problem does not exist. it is the device that is sending the 0 that rules the bus. You cannot disturb the other CPU's transmission because : if you can't raise one of the lines you backoff , and if it is the other one that can't raise one of the lines : he has to back-off.

This backoff condition will only occurr the moment that the 2 transmitted levels are not the same .

A graphical explanation tells more then a thousand words :



The 2 cpu's are accessing a slive in write mode at adress `1111001` . The slave ACK's this .

Then CPU1 wants to transmit `01010101` to the slave while CPU 2 wants to transmit `01100110` to the slave.The moment that the databits do not match anymore (what the cpu sends is different then wat is present on the bus.) One of them has to loose arbitration. Obviously this is the cpu which did not get his data on the bus. For as long as there has been no STOP present on the bus he won't touch the bus and leave the SDA and SCL line high .(yellow zone).

The moment a STOP was detected CPU2 can attempt to transmit again.

So from the above story we can conclude that is the one that is pulling the line LOW that always wins.The One wich wanted the line to be HIGH when it is beeing pulled low by the other looses the BUS .We call this a loss of arbitration or a BACKOFF condition

When the cpu gets a BACKOFF situation then it has to wait for a STOP condition to appear on the bus.Then it knows that the transmission has been completed .

If you struggled trough all of this , and the previous chapters then you are ready to face the world of I2C. !


next

The I2C FAQ 2.0

Synchronisation on the bus

The i2c protocol also includes a synchronisation mechanism. This can be used between slow and fast devices or between masters in a multimaster session . However to my knowledge there are currently no chips that use this mechanism. This must be implemented in software.

When a slow slave is attached to the bus then problems may rise.

Consider a serial EEPROM

(This is hypotetical !. Real devices use a different way to accomplish wat will be described below).

The actual writing process inside the eeprom might take some time. Now if you send multiple byte to such a device then the risk exists that you send it new data before it has completed the write cycle. this would corrupt the data or cause data-loss.

The slave must have some means to tell the master that it is busy inside . It could of course simple not respond to the ACK cycle. This would cause the master to send a stop and retry. (That's how it is done in real eeprom's.)

Other cases might not be so simple. Suppose an A/D convertoe. It might take some time for the conversion to complete. If the master would just go on it would be reading the result of the previous conversion instead of the new acquired data.

Now the synchronisation meachanism can come in handy. This mechanism works on the SCL line only.The Slave who want's it master to wait simply pulls the pulls the SCL low as long as needed.

The master is then not able of giving the ACK clockpulse because it cannot raise the SCL line . Of course the master software must check this condition and act appropriatly. In this case the master simply waits until it can raise the SCL line and then just goes on with whatever it was doing..

This is the routine the master must execute for a synchronisation aware Wait_for_ack sequence

While this is a very neat technique it is not beeing used (not to my knowledge that is). However it is enclosed in the I2C bus specification so you can perfectly use it .

There are a number of minor drawbacks involved when implementing this. If the SCL gets stuck due to an electrical failure of a circuit the master can go into deadlock.Of course this can be handled by timeout counters. Also you could hardly see this as a drawback , because , if the bus get's stuck then it's done communicating anayway.

Another drawback is speed . The BUS is locked at that moment. If you have rather long delays (long conversion time in our example above) then this penalizes the total bus speed a lot. Other devices cannot use the bus at that time either.

This technique does not interfere with the previous arbitration mechanism. because this will lead to backoff situations in other devices which possibly would want to 'claim' the bus. So there is no real drawback to this technique except the 'loss of speed' and some software overhead in the Masters.

You can use this mechanism between masters in a multimaster environment. this can prevent other master from taking over the bus . In a 2 master system this is not handy. But the moment you get 3 or more masters this is very usefull. A third master cannot interrupt a transfer between master 1 and 2 in this way. For some mission critical situations this can be very handy.

You can make this technique rigid by not pulling only the SCL line low , but also the SDA line. Then any master which is not in the 2 masters talking to each other will immediately Back off. Before you continue you first mke SDA back high ,and then SCL and go on. Any master which attempted to communicate in the mean time would have detected a BACKOFF situation and would be waiting for a STOP to appear.

The I2C FAQ 2.0 Special adresses and exceptions

Somewhere in the beginnig of this FAQ we have already mentioned that there are so called 'reserved adresses' This section details a bit more on these adresses and what they do. For information about the Extended adressing mode you must consult the next section.

Addess	R/W	Designation
0000-000	0	General Call address. (note 1)
0000-000	1	START byte (note 2)
0000-001	x	reserved for the (now obsolete) C-Bus format (note 3)
0000-010	x	Reserved for a different bus format (note 4)
0000-011	x	Reserved for future purposes (note 5)
0000-1xx	x	Reserved for future purposes
1111-1xx	x	Reserved for future purposes
1111-0xx	x	10 Bit slave adressing mode (note 6)

Note 1: The general call address.

This address is beeing used to access all devices on the bus which are capable of handling this and need this data . Devices capable of handling this call which do not need it will not answer.

All bytes transferred after this address may ar may not be taken in by the slaves who are responding to it. the moment that no slave is acknowledging a transmitted byte the operations is stopped by issuing a STOP on the bus.

The meaning of the general call adress is specified in the 1st byte transmitted after this ' general call '.

This first byte can contain the following information :

When the LSB is set to 0

0000-0110	Reset and write programmable part of slave address. All the devices who respond to this will reset and take in the programmable part of their adress. This is done by re-reading the levels on the adress select pins of the device (if any). this command can be used to reset an entire I2C system
0000-0100	The same as above but without the reset . This can be usefull if something is capable of changing the state of the adress select pins of a device. That way the device adress will move.

When the LSB is set to 1

xxxx-xxx1	This is a Hardware Call. If a device needs urgent attention from a master device without knowing which master it needs to send to it can use this call. This is a call 'to whom belongs ..' The device then embeds itt's own adress into the message. This call means as much as : please contact me i need to be serviced. All masters will listen and the master that knows how to handle the device with the adress transmitted will contact it's slave and act appropriatly. Currently no devices support this.

Note 2) The START adress.

This can be used between masters. A master which does not have an I2C interface in hardware but in software needs to monitor the bus all the time. since this can eat up a lot of processing time the START adress was invented. The masters can sample the bus at a low rate. The moment they detect that the SDA line is low (it is held low for over 7 clock periods) it can switch to a higher sampling rate to detect the Start condition.

This adress is not followed by a stop condition but by a repeated start.

Note 3 and 4

These adresses are used when other then I2C data has to be transmitted over the bus.

Note 5

These adresses are for further expantion and are currently not allowed

Note 6

This will be detailed on in the next section

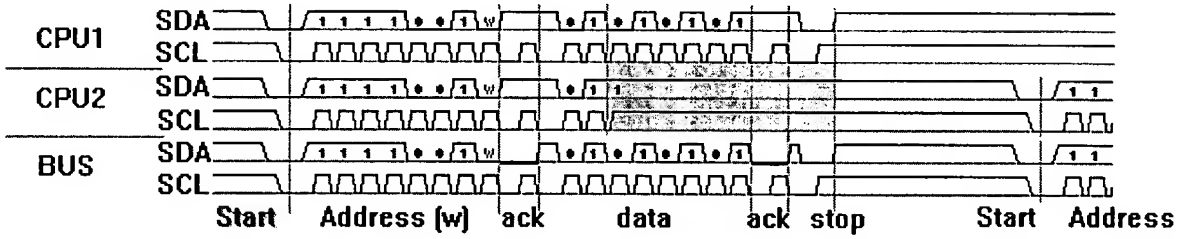
The I2C FAQ 2.0

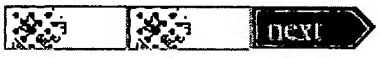
Electrical specs

These are the electrical specifications as set forth by Philips

Parameter	Symbol	Standard mode		Fast Mode		Unit
		min	max	min	max	
LOW level input voltage fixed input levels Vdd related input levels	Vil	 -0.5 -0.5	 1.5 0.3*Vdd	 -0.5 -0.5	 1.5 0.3*vdd	Volt
HIGH level input voltage fixed input levels Vdd related input levels	Vih	 3.0 0.7*Vdd	 Vddmax+0.5 Vddmax+0.5	 3.0 0.7*Vdd	 Vddmax+0.5 Vddmax+0.5	Volt
Hysteresis is Smitt trigger inputs fixed input levels Vdd related input levels	Vhys	 - -	 - -	 0.2 0.05Vdd	 - -	Volt
Pulse width of spikes which must be suppressed by the input filter	tsp	-	-	0	50	nS
Low level output voltage (open drain or open collector) at 3ma Sink current at 6 mA sink current	 Vol1 Vol2	 0 -	 0.4 -	 0 0	 0.4 0.6	Volt
Output fall time with a load <400pF with up to 3mA sink current with up to 6mA sink current	Tol	 - -	 250 -	 20+0.1Cb(2) 20+0.1Cb(2)	 250 250(3)	nS
Input current for an						

I/O pin vor an input level between 0.4V and 0.9*Vddmax	Ii	-10	10	-10(4)	10(4)	uA
Capacitance for each I/O pin	Ci	-	10	-	10	pF
SCL Clock frequency	fSCL	0	100	0	400	kHz
Bus free time between STOP and START	tBUF	4.7	-	1.3	-	uS
Hold time after START	tHD:STA	4.0	-	0.6	-	uS
Low period of SCL	tLOW	4.7	-	1.3	-	uS
High period of SCL	tHIGH	4.0	-	0.6	-	uS
Setup for a repeated start	tSU:STA	4.7	-	0.6	-	uS
Data hold time	tHD:DAT	0	-	0	0.9	uS
Data Set-up time	tSU:DAT	250	-	100	-	nS
Rise time of both SDA and SCL	tr	-	1000	20+0.1Cb	300	nS
Fall time of both SCL and SDA	tf	-	300	20+0.1Cb	300	nS
Set-up time for STOP condition	tSU:STO	4.0	-	0.6	-	uS
Capacitive load for each bus line	Cb	-	400	-	400	pF





The I2C FAQ 2.0

Enhanced I2C (fast mode)

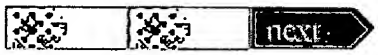
Since the first I2C spec release (which dates back from 1982) a couple of improvements have been made.In 1993 the new I2C spec was released.This new spec conatains some additional sections covering FAST mode and 10 -Bit addressing. In this section the Fast mode will be covered , while in the next section information about 10 bit addressing is given.

In the FAST mode the physical bus parameters are not altered. The protocol,Bus levels,Capacitive load etc.. remain unchanged. However the datarate has been increased to 400 Kbit/s and a constraint has been set on the level of noise that can be present in the system. To accomplisch this task a number of changes have been made to timing.

Since all CBUS activities have been canceled ,there is no compatibility anymore with CBUS timing.The development of IC with CBUS interface has been stopped. The existing CBUS ic's are being taken out of production.Furthermore the CBUS devices cannot handle these higher clock rates.

The input of the FAST mode devices all include Schmitt triggers to suppress noise.The output buffers include slope control for the falling edges of the SDA and SCL signals.If the power supply of a FAST mode device is switched off the BUS pins must be floating so that they do not obstruct the bus.

The pullup resitor must be adapted. For loads up to 200 pf a resistor is sufficient.For loads between 200pf and 400pF a current source (active pull-up) is preferred.

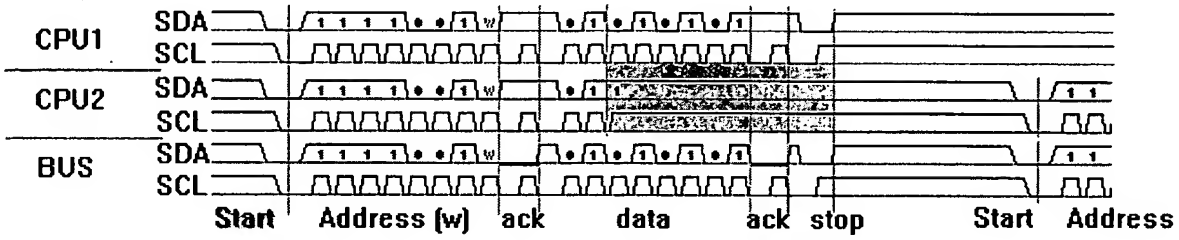


The I2C FAQ 2.0

Extended adressing

Due to the increasing popularity of the I2C bus the address space is nearly exhausted.This starts posing problems for people currently in the phase of designing a new I2C compatible IC. Therefore the I2C standard has been adapted.

A chip that conforms to the new standard receives 2 address bytes. The first consists of the extended adressing reserved address including the 2MSB's of the device address and the Read/Write bit. The second byte contains the LSB's of the address.



This scheme insures that the 0 bit addressing mode stays completely transparent for the other devices on the bus. Normally any new design should adept to this new addressing scheme.